hyväksymispäivä arvosana

arvostelija

Progressiivisten web-sovellusten kehittäminen

Niki Ahlskog

Helsinki 29.4.2019

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Osasto — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Niki Ahlskog

Työn nimi — Arbetets titel — Title

Progressiivisten web-sovellusten kehittäminen

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	29.4.2019	58 sivua + 0 liitesivua

Tiivistelmä — Referat — Abstract

Progressiivisen web-sovelluksen (Progressive Web Application, PWA) tarkoitus on hämärtää tai jopa poistaa raja sovelluskaupasta ladattavan sovelluksen ja normaalin verkkosivuston välillä. PWA on kuin mikä tahansa normaali verkkosivusto, mutta se täyttää lisäksi seuraavat mittapuut: Sovellus skaalautuu mille tahansa laitteelle. Sovellus tarjotaan salatun yhteyden yli. Sovellus on mahdollista asentaa puhelimen kotinäytölle pikakuvakkeeksi, jolloin sovellus avautuu ilman selaimesta tuttuja navigointityökaluja ja lisäksi sovelluksen voi myös avata ilman verkkoyhteyttä.

Tässä työssä käydään läpi PWA-sovelluksen rakennustekniikoita ja määritellään milloin sovellus on PWA-sovellus. Työssä mitataan PWA-sovelluksen nopeutta Service Workerin välimuistitallennusominaisuuksien ollessa käytössä ja ilman. PWA-sovelluksen luomista ja käyttöönottoa tarkastellaan olemassa olevassa yksityisessä asiakasprojektissa. Projektin tarkastelussa kiinnitetään huomiota PWA-sovelluksen tuomiin etuihin ja kipupisteisiin.

Tuloksen arvioimiseksi otetaan Google Chromen Lighthouse -työkalua käyttäen mittaukset sovelluksen progressiivisuudesta ja nopeudesta. Lisäksi sovellusta vasten ajetaan Puppeteer-kirjastoa hyödyntäen latausnopeuden laskeva testi useita kertoja sekä tarkastellaan PWA-sovelluksen Service Workerin välimuistin hyödyllisyyttä suorituskyvyn ja latausajan kannalta. Jotta Service Workerin välimuistin käytöstä voidaan tehdä johtopäätökset, nopeuden muutosta tarkastellaan progressiivisten ominaisuuksien ollessa käytössä ja niiden ollessa pois päältä. Lisäksi tarkastellaan Googlen tapaustutkimuksen kautta Service Workerin vaikutuksia sovelluksen nopeuteen.

Testitulokset osoittavat että Service Workerin välimuistin hyödyntäminen on nopeampaa kaikissa tapauksissa. Service Workerin välimuisti on nopeampi kuin selaimen oma välimuisti. Service Worker voi myös olla pysähtynyt ja odotustilassa käyttäjän selaimessa. Silti Service Workerin aktivoiminen ja välimuistin käyttäminen on nopeampaa kuin selaimen välimuistista tai suoraan verkosta lataaminen.

ACM Computing Classification System (CCS):
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords
PWA, progressiiviset web-sovellukset, mobiili selainsovellus, progressiivisuus

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Mobiilisovellukset ovat jatkuvasti kehittyvää ja muuttuvaa teknologiaa, kuten myös strategiat niiden kehittämiseksi. Tämän päivän mobiilisovelluksia kehitettäessä on teknologiavaihtoehdoissa useita vaihtoehtoja. Kehittäjät voivat valita ainakin neljästä eri mallista: mobiili-web-sovellus (verkkosivu joka on toteuttu myös mobiililaitteella toimivaksi), natiivisovellus (ohjelmoitu suoraan laitteen natiivikoodilla), hybridisovellus (verkkosovellus joka on paketoitu sovellukseksi), tai uusimpana teknologiana progressiivinen web-sovellus (Progressive Web App, PWA). Jokaisessa vaihtoehdossa on etuja ja haittoja. Web-tekniikoihin panostaminen näyttäisi kuitenkin olevan kustannustehokasta, sillä nykyiset sovelluskaupat on jaettu eri laitealustojen kesken. Googlen julkaiseman tiedotteen mukaan [1] PWA-sovellus käyttää moderneja web-tekniikoita saavuttaakseen sovelluksen kaltaisen käytöksen. Kun PWA-sovelluskehitykseen on tutustunut, natiivisovelluskehitystä ei enää välttämättä tarvita. PWA-sovellukset käyttäytyvät kuin natiivisovellukset, eikä niiden suorituskyvyssä välttämättä huomaa eroa natiivisovellukseen verrattuna. Lisäksi PWA-sovellusten kehittäminen on nopeampaa ja halvempaa yhden ainoan koodirungon takia.

Mobiili verkkosivu itsessään muodostuu sovellukseksi käyttäen verkon parhaita tekniikoita. Tämä tarkoittaa sitä, että verkkosivu voidaan asentaa pikakuvakkeeksi mobiililaitteen sovellusvalikkoon, jonka jälkeen sovellus on mahdollista avata samaan tapaan kuin natiivisovellus. Lisäksi sovelluksessa on mahdollista käyttää sellaisia toimintoja jotka olivat ennen mahdottomia. Esimerkiksi lähettää sovelluksen asentaneille käyttäjille sovellusilmoituksia ja tallettaa sovelluksen tietoja käyttäjän laitteen välimuistiin. PWA-sovellus on siis tavallinen verkkosivu, joka hyödyntää selainten uusimpia ominaisuuksia.

PWA-termin määritteli Google Chromen insinööri Alex Russel [2, 3] omassa blogikirjoituksessaan vuonna 2015 yhdessä suunnittelija Frances Berrimanin kanssa [4]. PWA-sovellus voidaan määritellä Alex Russelin sanoin normaaliksi verkkosivuksi, jota on paranneltu hyödyntämällä selaimen uusimpia ominaisuuksia. Yahoo Developer-portaalissa tehdyn tutkimuksen mukaan [5] 90 % mobiililaitteilla vietetystä ajasta käytetään sovellusten parissa ja vain 10 % ajasta mobiililaitteilla käytetään verkon selaamiseen.

Alustariippumattomassa sovelluskehityksessä etuna on, että iOS- ja Android-alustoille ei tarvitse erikseen kehittää omaa sovellusta ja ylläpitää kahta eri lähdekoodia [6]. Yksi ja sama ohjelma riittää kummallekin alustalle. PWA-sovelluksia ei tarvitse paketoida, eikä jaella sovelluskaupan kautta, sillä PWA-sovellus on verkkosivu, tai web-selaimella toimiva sovellus, joka muistuttaa mahdollisimman paljon natiivia mobiili-, tai työpöytäsovellusta ja joka on mahdollista asentaa pikakuvakkeeksi käyttäjän laitteen sovellusvalikkoon. Perimmäisenä ajatuksena on hämärtää verkkosovellusten ja natiivisovellusten rajaa niin että käyttäjä ei tunnista eroa PWA-sovelluksen ja natiivisovelluksen välillä.

Alex Russelin ajatuksena [2] PWA-sovelluksissa on ollut niiden muotoutuminen sovellukseksi ilman, että käyttäjän tulee heti tehdä päätös sovelluksen asentamisesta ja

käyttölupien luovuttamisesta. Esimerkiksi selaimista tuttu lupien kysyminen pysyy ennallaan. Sovellus kysyy mikrofonin, kameran, tai minkä tahansa muun ominaisuuden käyttölupaa vasta kun toimintoa tarvitaan ensimmäisen kerran. Muutaman käyttökerran jälkeen sovellus kysyy käyttäjältä, halutaanko puhelimen työpöydälle asentaa pikakuvake sovellukseen. Jos sovellus lähettää sovellusilmoituksia, nekin voidaan joko hyväksyä tai hylätä käyttäjän toimesta. Tämä siis tarkoittaa, että käytön myötä verkkosivu muuttuu käyttäjän puhelimessa natiivin mobiilisovelluksen kaltaiseksi ilman suurempaa sitoutumista sovelluksen asentamiseen ja kaikkien käyttölupien luovuttamiseen heti.

PWA-sovellukset eivät kuitenkaan sovellu sellaisiin käyttötarkoituksiin, joissa täytyy päästä käsiksi puhelimen rajapintoihin, kuten vaikkapa kalenteriin tai osoitekirjaan. PWA-sovelluksen valintaan vaikuttaa siis sovelluksen käyttötarkoitus.

Tässä tutkielmassa selvitetään millaisia vaikutuksia sovelluksen nopeuteen progressiivisuudella saavutetaan ja kuinka helppoa työkalujen käyttöönotto on. Esimerkkiprojektissa mitataan sovelluksen nopeutta välimuistitallennuksen ollessa käytössä sekä ilman. Mittarina käytetään Google Chrome -selaimen kehittäjätyökaluissa toimitettavaa Lighthouse-työkalua. Lighthouse-metriikoiden lisäksi sovellusta vasten suoritetaan latausajan laskeva automatisoitu testi, jonka perusteella voidaan kuvata sovelluksen latausajat keskiarvoineen.

Työ on jaettu seitsemään päälukuun. Luvussa 2 esitellään PWA-sovelluksen vaatimukset, toteutustekniikoita ja selvennetään termin määritelmää. Luvussa 3 tarkastellaan muita mobiilikehitysteknologioita. Luvussa 4 esitellään testattava sovellus ja tutkimusongelma. Luvussa 5 esitellään sovelluksen tekninen toteutus ja projektiin asennetaan tarvittavat työkalut PWA-sovelluksen kehittämiseksi. Luvussa 6 suoritetaan mittaukset käyttämällä Google Chrome -selaimen Lighthouse työkalua. Nopeusmittaukset Service Workerin välimuistin käytöstä suoritetaan Puppeteer-kirjastolla. Lisäksi luvussa 6 tarkastellaan kuinka paljon vaikutuksia Service Workerin välimuistilla oli sovelluksen nopeuteen. Luvussa 7 tehdään yhteenveto tutkimuksesta.

2 PWA-sovellusten esittely

Tässä luvussa esitellään PWA-teknologia ja sen vaatimukset. Luvussa esitellään miksi PWA-sovelluksia tarvitaan.

2.1 PWA-sovelluksen ominaisuudet

Vuonna 2014 verkon käyttö mobiililaitteilla ohitti työasemat [4]. Tämä osoittaa, että verkkosovellusten tekeminen mobiililaitteille sopiviksi on tärkeää. Useissa tapauksissa yrityksen täytyy kehittää erikseen natiivisovellus kullekin laitealustalle ja ohjelmoida se käyttäen laitealustalle tarkoitettua ohjelmointikieltä. Tällöin ongelmaksi muodostuu usein resurssien rajallisuus: kehittäjiä ei ole tai sovelluskehitys maksaa liikaa. Yrityksellä, jolla ei ole resursseja tai haluja palkata erikseen mobiilikehittäjiä on alustariippumattomasta kehittämisestä tullut suosittua [3]. Alustariippumaton kehittäminen tarkoittaa yhden sovelluksen kehittämistä usealle laitteelle yleensä perustuen web-teknologioihin [7]. Sovelluskehityksen aika ja kehityskustannukset pienenevät, kun sovellusta ei tarvitse kehittää erikseen jokaiselle laitealustalle. Näin tuote saadaan nopeammin markkinoille.

PWA on kokoelma verkon parhaita käytäntöjä [8] ja PWA-sovellukset pyrkivät toimimaan natiivisovelluksen kaltaisesti. Ihannetilanne on sellainen, jossa käyttäjä ei pysty sanomaan, onko sovellus natiivisovellus vai verkkosovellus. PWA-sovelluksen tarkoitus on hämärtää natiivisovelluksen ja web-sovelluksen rajaa. Rajaa halutaan hämärtää siksi, että yleensä natiivisovellusten julkaiseminen sovelluskauppoihin on hidasta ja aikaa vievää. Hyväksymisprosessit saattavat kestää sovelluskaupoissa pitkään ennen sovelluksen julkaisua. Lisäksi sovellusten julkaisu on maksullista. iOS kaupan lisenssi maksaa 99 dollaria vuodessa yhdelle kehittäjälle ja yrityksille 299 dollaria. Android-alustan Play-Storen lisenssi on kertakustanteinen ja maksaa 25 dollaria yhdelle kehittäjälle. Kehittäjän ja tilaajan näkökulmasta PWA-sovelluksen kehittäminen on siis järkevää. Kehittäjän ei tarvitse osata jokaisen laitealustan ohjelmointikieltä, eli sovellusta ei tarvitse ohjelmoida usealle eri mobiilialustalle [9]. Tilaajalle sovelluksen kehittäminen on halvempaa kun yksi ja sama kehittäjä riittää kummallekin laitealustalle. Myös sovelluksen julkaisusykli on nopeampi sillä sovellusta ei tarvitse julkaista ja hyväksyttää sovelluskaupoissa. Sovellus julkaistaan suoraan verkossa. Sovellus muotoutuu käyttäjän laitteelle alustasta riippumatta ja yksi ainoa versio riittää.

Alex Russell määritteli blogikirjoituksessaan [2] millainen PWA-sovelluksen tulee olla:

- **Responsiivinen:** Sovelluksen tulee sopia mille tahansa näytölle koosta ja resoluutiosta riippumatta.
- **Yhteyksistä riippumaton:** Sovelluksen tulee käynnistyä myös verkkoyhteydettömässä tilassa.

- **Natiivisovelluksen kaltainen:** Sovelluksen tulee näyttää ja toimia natiivisovelluksen tavoin.
- **Tuore:** Sovelluksen pitää hakea aina verkkoyhteyden ollessa päällä uusin versio käyttäjän huomaamatta.
- **Turvallinen:** Sovellus pitää aina tarjota salattuna Transport Layer Security -protokollaa (TLS) käyttäen.
- **Löydettävä:** Sovelluksen tulee noudattaa W3C-yhteisön määrityksiä ja tarjota manifesti, joka määrittää sovelluksen. Sovellus voidaan löytää tavallisista hakukoneista sovelluskauppojen sijaan. Mikäli sovellus täyttää kaikki PWA-sovelluksen kriteerit, manifestistä poimitaan sovelluksen nimi ja kuvake, jos käyttäjä päättää asentaa sovelluksen laitteelleen.
- **Sitouttava:** Sovelluksen tulee olla mahdollista muistuttaa olemassaolostaan esimerkiksi sovellusilmoituksilla ja näin saada käyttäjä palaamaan sovelluksen pariin.
- **Asennettava:** Sovelluksen voi asentaa mobiililaitteen sovellusvalikkoon pikakuvakkeeksi, jolloin se on nopeasti saatavilla.

2.2 Miksi PWA-sovelluksia tarvitaan

Natiivi- ja web-sovelluksissa on monenlaisia haasteita. Yksi haasteista on Internetin nopeus. Internet ei ole kaikkialla maailmassa nopea. Noin 60 % maailmasta käyttää yhä 2G-verkkoa [8] ja paikoitellen verkko on niin ruuhkautunut, että sen tiedonsiirtokyky ei ole riittävä. Verkkosivun tulee latautua tarpeeksi nopeasti. Yhä useammin sivuilla on erilaisia bannereita, multimediaa ja moderneja animaatioita. Tutkimuksen mukaan 53 % käyttäjistä hylkää sivun, mikäli se latautuu ja käyttäytyy liian hitaasti [8].

Toinen haaste on saada käyttäjä asentamaan sovellus omalle laitteellen. Kapoorin tutkimuksen mukaan [8] kynnys asentaa sovellus omalle laitteelle on korkea, sillä asennettu sovellus vie tilaa mobiililaitteelta ja sovellukselle pitää antaa lisäksi täydet käyttöoikeudet sen tarvitsemiin rajapintoihin. ComScoren mukaan keskimääräinen käyttäjä lataa ja asentaa laitteellensa nolla sovellusta kuukaudessa [10].

Kolmas haaste on käyttäjän sitouttaminen. Mobiilikäyttäjät viettävät suurimman osan ajastaan natiivisovelluksissa. Noin 80 % ajasta käytetään kolmen tärkeimmän natiivisovelluksen parissa. Verkkosivujen käyttöaste on noin viidesosan sovelluksiin verrattuna. Näin ollen suurin osa käyttäjistä ei ole sitoutettu palveluun [8]. Väitteen esittäjältä, Kapoorilta, pyydettiin tarkennusta asiaan 15.3.2019 saamatta vastausta.

PWA-sovellukset pyrkivät ratkaisemaan näitä ongelmia seuraavilla keinoilla.

- **Nopeus:** PWA tarjoaa Service Workerin avulla mahdollisuuden tallentaa tietoa välimuistiin, joka mahdollistaa sovelluksen latautumisen nopeammin käyttäjän laitteella. Sovelluksen käynnistäminen on siis erityisen nopeaa, vaikka verkkoon ei olisi vielä edes tehty kutsuja.

- **Saumaton käyttökokemus:** PWA-sovellus tuntuu ja käyttäytyy kuin natiivisovellus. PWA-sovellus on mahdollista asentaa mobiililaitteen kotinäytölle. Erona natiivisovellukseen on se, että sovellusta voi käyttää myös ilman, että tekee asennuspäätöksen. Sovellusta ei siis ole pakko asentaa. PWA-sovellus tarjoaa kuitenkin vaihtoehdon lisätä pikakuvake sovellusvalikkoon. PWA-sovellus voi lähettää ilmoituksia (push notifications) ja sovellus voi myös hyödyntää laitteen rajapintoja, kuten kameraa, tai mikrofonia.

- **Luotettava käyttökokemus:** Sovellus voidaan avata myös silloin, kun verkkoyhteyttä ei ole saatavilla. Välimuistiin tallennetun tiedon perusteella viimeisimmät muistissa olevat asiat saadaan esitettyä.

- **Sitouttava:** Kun sovellukseen voidaan lähettää ilmoituksia, voidaan käyttäjää sitouttaa sovellukseen lähettämällä muistutuksia, tai tärkeitä tietoja.

Tal Aterin julkaisemassa kirjassa "Building Progressive Web Apps"[11] viitataan vuoden 2016 comScoren raporttiin [12], jonka mukaan keskiverto henkilö viettää 80-90 % ajastaan käyttäen vain viittä tärkeintä sovellusta. Samasta raportista selviää myös, että on paljon helpompaa tavoittaa suurempi yleisö verkkosivulla, kuin natiivisovelluksella.

Jotta käyttäjän saisi asentamaan mobiilisovelluksen, täytyy sovelluksesta ensin saada tietää jostain. Sen jälkeen täytyy etsiä kyseinen sovellus sovelluskaupasta, ladata ja asentaa se. Sovellukselle täytyy antaa tarvittavat käyttöoikeudet heti. Viimeiseksi käyttäjä voi vasta avata sovelluksen ja ehkä jopa tehdä sillä jotain. Näiden toimenpiteiden suorittaminen ei kuulosta huonolta, jos kyseessä on jokin käyttäjän pitämä ja luottama sovellus, kuten esimerkiksi Facebook tai Twitter. Aterin kirjan mukaan [11] on useita tutkimuksia jotka todistavat, että keskimäärin 20 % käyttäjistä ei suorita toimenpiteitä loppuun ja jättää sovelluksen asennuksen kesken.

Kun käyttäjä vierailee haluamansa sovelluksen sivulla, ensimmäinen asia mikä käyttäjälle näytetään, on mainos uudesta mobiilisovelluksesta. Esimerkiksi, jos käyttäjä haluaa tarkistaa päivän sään joltakin verkkosivulta, tulisi eteen koko ruudun kokoinen mainos sääsovelluksesta ja linkki sovelluskauppaan. Tämä peittää varsinaisen sisällön, josta käyttäjä on kiinnostunut, vaikka tieto olisi heti saatavilla bannerin alla.

PWA-sovellukset ovat kääntämässä suunnan. Käyttäjä voi alkaa käyttämään palvelua heti ja sovellus pyytää lisäämään pikakuvakkeen käyttäjän työpöydälle mikäli hän vierailee sivustolla useamman kerran. Myös käyttöoikeuksia lisätään progressiivisesti sitä mukaa kun niitä tarvitaan. PWA-sovellus siis poistaa tarpeen mainostaa erikseen mobiilisovelluksia. Käyttäjiä ei tarvitse enää ohjata sovelluskauppoihin, sillä verkkosivusto alkaa itse ehdottaa pikakuvakkeen luomista käyttäjälle.

PWAstats [13] on julkaissut tilastoja, miten yritysten nykyisten sivujen muuttaminen PWA-sovellukseksi on tuottanut lisätuloja tai käyttäjiä. Esimerkiksi Uberin PWA-sovellus on nopea jopa 2G-verkossa. Uberin PWA-sovellus latautuu 2G-verkossa alle kolmessa sekunnissa, jos sovellus on jo valmiiksi välimuistissa. Suosittu

deittipalvelu Tinder taas pienensi latausaikaa vajaasta 12 sekunnista alle viiteen sekuntiin, joka vaikutti suoraan käyttäjäkokemukseen ja käyttäjän sitoutumiseen. Tutkimuksen mukaan [14] Tinderin käyttäjät pyyhkäisivät, viestittelivät ja muokkasivat tietojaan useammin PWA-sovelluksessa kuin natiivisovelluksessa. PWA-sovellusten pitäisi siis toimia nopeammin kuin tavallisten verkkosivustojen.

2.3 PWA-sovelluksen vaatimukset

Google on julkaissut PWA:lle tarkastuslistan, jota noudattamalla sovelluksesta saadaan progressiivinen [15]. Periaatteessa mikä tahansa verkkosivusto voidaan tehdä progressiiviseksi sovellukseksi [16] riippumatta siitä, millä ohjelmistokehyksellä se on ohjelmoitu [1], kunhan vain PWA-sovelluksen ehdot täyttyvät.

Web App Manifest on sovelluksen juureen määritettävä manifest.json-tiedosto, jolla projektista saadaan sovelluksen kaltainen. Tiedosto voi pitää sisällään esimerkiksi tiedot seuraavista asioista:

- sovelluksen nimi ja lyhytnimi,
- sovelluksen teeman väri,
- taustaväri,
- sovelluksen oletus käyttöasento, joko pysty-, tai vaaka-asento,
- aloitustiedosto, yleensä index.html tai index.js,
- ikonitiedot eri resoluutioissa,
- aloitusruutu.

Toteutuksen kannalta kyseessä on JSON-tiedosto, joka pitää sisällään metatietoa sovelluksesta. Kun selain havaitsee manifest.json-tiedoston, se tietää että kyseessä on sovelluksen kaltainen verkkosivu. Tiedostossa määritelty kuvake on se, joka luodaan käyttäjän kotinäytölle asennuspäätöstä tehtäessä. Kun sovellus avataan kuvakkeesta, muuttuu mobiililaitteen tilapalkki sovelluksen teeman väriä vastaavaksi, mikäli sellainen on määritelty. Manifestin tarkoitus on tarjota muutettavia asetuksia kehittäjille yhdessä konfiguraatiotiedostossa [3]. Manifestia voidaan siis käyttää määrittämään sovelluksen käyttäytyminen ja tyyli. Kun sovelluskuvaketta klikataan mobiililaitteen sovellusvalikosta, kestää selaimen käynnistymisessä pieni hetki [16]. Tänä aikana näytetään sovelluksesta aloitusruutu, joka generoituu automaattisesti. Taustavärinä on sovelluksen väri ja keskellä sovelluksen ikoni. Nämä ovat manifestissa määriteltyjä tietoja. Aloitusruudun näyttämisellä sovelluksen käytöstä saadaan parempi käyttökokemus [16]. Käyttäjälle näytetään sovelluksesta heti ainakin jotain

[1] AngularJs (https://angularjs.org/), React (https://reactjs.org/), VueJs (https://vuejs.org/), Wordpress (https://wordpress.org/), Drupal (http://www.drupal.fi/)

varsinaisen käyttöliittymän latautuessa taustalla. Tästä tulee käyttäjälle tunne, että sovellus toimii nopeasti.

Manifestissa määritelty nimi on sovelluksen nimi mobiililaitteen valikossa. Lyhytnimitietoa käytetään silloin, kun sovellusvalikon näkymä on pieni, esimerkiksi matkapuhelimissa.

Aloitustiedosto on usein index.html tai index.js. Joissain tapauksissa voi olla hyödyllistä ohjata käyttäjä sivulle, jossa pystyy tekemään jotakin. Esimerkiksi sellaisessa tapauksessa, missä indexisivu on sisäänheittosivu tai sovelluksen mainossivu. Tässä tapauksessa voi olla järkevää ohjata käyttäjä vaikka sovelluksen kirjautumisikkunaan.

Sovelluskuvake tarvitaan kun käyttäjä tekee päätöksen asentaa sovellus puhelimeensa. Käytettävälle laitteelle luodaan manifest.json tiedostossa määritelty sovelluskuvake joko työpöydälle tai sovellusvalikkoon. Sovelluskuvakkeeksi pitäisi käydä useampi eri kuvaformaatti, kunhan vain tiedostomuoto on määritelty erikseen. Sovelluskuvakkeita pitää olla lisäksi useita eri kokoja, jotta resoluutio riittää kaikenkokoisille näytöille.

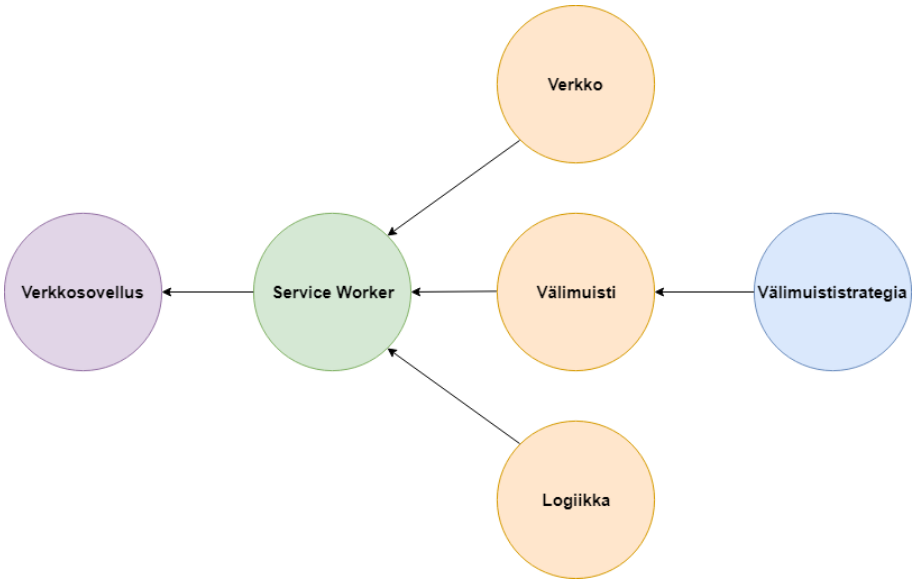
Service Worker on teknologia, joka on koko PWA-sovelluksen pohja. Service Worker on uusi rajapinta verkkoselaimissa. Service Worker on kokoelma erilaisia rajapintoja [17]. Service Worker toimii sovelluksen ja verkon välissä käyttäjän laitteella välittäjänä (proxy) kuva 2.1.

Service Workerin päällimmäinen tarkoitus on toimia välimuistina. Service Workerilla voidaan kopioida verkosta haetut tiedot välimuistiin. Seuraavalla kerralla, kun käyttäjä pyytää samoja tietoja, ne ladataankin puhelimen välimuistista. Websovelluksen, joka lataa välimuistiin kaiken tiedon sivustolta, pitäisi olla huomattavasti nopeampi kuin sovelluksen, joka ei hyödynnä välimuistia [18].

Service Workeriin voidaan ohjelmallisesti määrittää, mitä tiedostoja tallennetaan ja kuinka niitä käsitellään. Tapoja kutsutaan välimuististrategiaksi. Välimuististrategiassa voidaan käyttää esimerkiksi pelkkää välimuistia, tai pelkkää verkkoyhteyttä tiedostoa haettaessa. Mikäli pyydetty tiedosto on epäolennainen, esimerkiksi tekstityypit, eikä muutu kovin usein, voidaan pyynnöt ohjata aina välimuistiin. Välimuististrategiaan voidaan myös määrittää, että tekstityypit on kuitenkin päivitettävä kerran viikossa. Pyynnöt voidaan siis ohjata välimuististrategiasta riippuen joko välimuistiin, tai verkkoon sovelluksen palvelimelle. Service Workeria käyttämällä kehittäjä voi ohjelmallisesti määritellä välimuistiin ladattavat asiat, sekä ladata etukäteen välimuistiin määriteltyjä tiedostoja, kuten kuvia ja muuta ennalta tiedettyä dataa. Service Workeria hyödyntämällä sovellus voidaan avata myös Offline-tilassa ja näyttää pyydettävät tiedot laitteen välimuistista. Välimuististrategiassa voidaan määrittää esimerkiksi kuville, että ne haetaan aina välimuistista. Mikäli kuvaa ei löydy välimuistista se haetaankin verkosta. Välimuististrategioita ovat seuraavat [19]:

- Tiedosto haetaan pelkästään välimuistista.
- Tiedosto haetaan pelkästään verkosta.

- Tiedosto haetaan välimuistista, tai vaihtoehtoisesti verkosta, jos sitä ei löydy välimuistista.
- Tiedosto haetaan verkosta, tai vaihtoehtoisesti välimuistista, jos sitä ei löydy verkosta.
- Verkon ja välimuistin kilpailu. Joskus pienten tiedostojen hakeminen verkosta voi olla nopeampaa kuin hitaalta massamuistilta.
- Välimuisti ensin, sitten verkko. Hyödyllinen silloin, kun jokin tieto päivittyy useasti. Välimuistista voidaan nopeasti ladata vanha tieto ja taustalla päivittää uusi tieto verkosta.
- Yleinen varasuunnitelma. Kun tiedostoa ei löydy välimuistista eikä verkosta, näytetään tilalla vaihtoehtoinen kuva tai esimerkiksi teksti: "ei käytettävissä ilman verkkoyhteyttä".



Kuva 2.1: Service Workerin toimintalogiikka.

Service Workerin päällimmäiset hyödyt ovat [20]:

- Kaikki uudet pyynnöt voivat suoraan ohittaa verkon, mikäli pyydettävä tiedosto on tallennettu käyttäjän laitteen välimuistiin.
- Vanhat välimuistissa olevat asiat voidaan näyttää heti, sillä aikaa, kun uusia tietoja haetaan taustalla.
- Service Workerilla voidaan hakea vain osittaista tietoa sen sijaan, että haettaisiin kaikki mahdollinen.

Toinen Service Workerin käyttötapa liittyy notifikaatioihin. Koska Service Workeria suoritetaan verkkoselaimen taustapalveluna, vaikka varsinainen sovellus olisi suljettuna, voidaan Service Workerin toiminta herättää ilmoituspalvelun kautta. Jokainen web-selain hallitsee ilmoitukset oman ilmoituspalvelunsa kautta. Kun käyttäjä antaa selaimelle luvan lähettää ilmoituksia, voidaan sovellus silloin tilata selaimen ilmoituspalveluun [21]. Tämä luo objektin, joka sisältää päätepisteen (endpoint) tilaukselle, joka on erilainen jokaisessa selaimessa. Esimerkiksi Googlen ja Mozillan ilmoituspalvelut [1]. Lisäksi tilaukseen tarvitaan julkinen avain (Voluntary Application Server Identification for Web Push, VAPID). Ilmoitukset lähetetään selaimen tilausosoitteeseen salattuna julkisella avaimella. Ilmoituspalvelu lähettää lopulta ilmoituksen käyttäjän mobiililaitteelle. Näin ollen sovelluksessa voidaan näyttää ilmoituksia jopa silloin, kun sovellus itsessään on suljettuna. Näin ei ole aikaisemmin voinut tehdä verkkosovelluksissa.

Service Workerin standardi on määritelty vuonna 2009 W3C:n toimesta. Service Workeria suoritetaan rinnakkain JavaScriptin pääsäikeen (main thread) kanssa. Service Workerin elinkaari on seuraavanlainen. Kun käyttäjä vierailee verkkosivustolla ensimmäisen kerran, Service Workerin olemassaolo sekä selaimen tuki Service Workerille tarkistetaan. Jos sitä ei löydy se asennetaan. Service Worker rekisteröidään toimimaan sovelluksen tietyllä osa-alueella, tai koko sovelluksessa. Tietyissä tapauksissa Service Workerin käyttöä voi olla järkevää rajoittaa sovelluksen tietylle osa-alueelle, esimerkiksi pelkkään käyttäjän näkymään [16]. Rekisteröinnin, eli asentamisen jälkeen Service Worker on aktiivinen ja vastaa verkkokutsuihin.

Service Workerilla on kaksi tilaa, aktiivinen ja pysäytetty. Service Worker ei ole aina aktiivisena selaimessa vaan se menee lepotilaan kunnes sitä tarvitaan uudestaan. Service Worker aktivoituu uudestaan verkkopyynnöistä. Service Workerin elinkaari on havainnollistettu kuvassa 2.2.

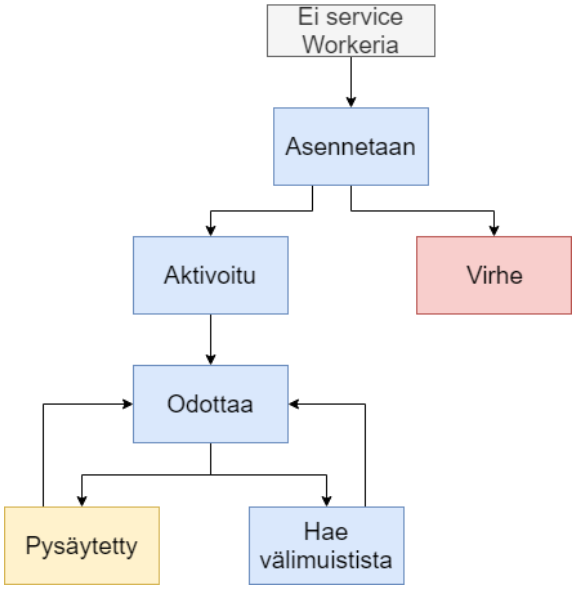
Service Workerin tuki eri selaimilla on kuvassa 2.3. Kuvasta nähdään, että Service Worker on pääosin tuettuna kaikissa yleisimmissä selaimissa. Joissain selaimissa Service Worker on kuitenkin vielä kokeellinen ominaisuus. Jos selaimessa ei vielä ole Service Worker tukea, täytyy Service Workerin rekisteröinti jättää tekemättä. Selaimen Service Workerin tuen tarkistaminen ja rekisteröinti tapahtuu listauksen 2.1 mukaisesti.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/service-worker.js');
  });
}
```

Listaus 2.1: Service Workerin tuen tarkistus ja rekisteröinti selaimessa.

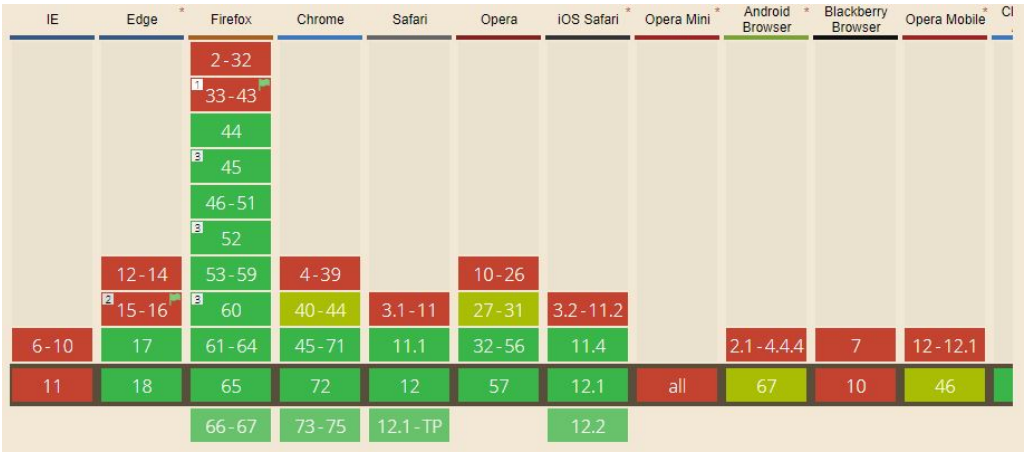
Jotta Service Worker saadaan asennettua ja aktiiviseksi, tulee selaimelle määritellä mistä tiedostosta Service Worker löytyy. Service Worker rajapinnan saatavuus

[1] https://fcm.googleapis.com, https://push.services.mozilla.com



Kuva 2.2: Service Workerin elinkaari yksinkertaistettuna.

tarkistetaan ehtolausekkeen sisällä. Mikäli tuki löytyy selaimesta, Service Worker rekisteröidään heti, kun sivu on latautunut.



Kuva 2.3: Service Workerin selaintuki [22].

Service Workerin tallennustila sovittiin Syksyllä 2017 Microsoft Edge Web -tapahtumassa. Selainvalmistajat sopivat keskenään selaimen tallennustilan koosta. Yleiskooksi sovittiin 50 megatavua aina yli 20 gigabitin tallennustilaan [23]. 20 gigatavua ei kuitenkaan päde aina. Nyrkkisääntönä on, että kovalevyn tilasta voidaan varata korkeintaan 4 % tai 20 gigatavua, kumpi vain on pienempi. Taulukossa 2.1 on esitetty tallennustilan varaamista erikokoisilla kiintolevyillä. Sivuston raja tarkoittaa samasta verkkotunnuksesta olevia sovelluksia. Mikäli saman verkkotunnuksen alta asennetaan useampi sovellus, on tallennustila yhteinen kaikille sen tunnuksen

sovelluksille.

Service Workerin tallennustrategioita miettiessä on syytä kiinnittää huomiota tallennustilan rajallisuuteen. Kaikkea tietoa ei välttämättä ole järkevää tallettaa käyttäjän laitteelle. Esimerkiksi Apple on päättänyt rajoittaa PWA-sovellusten välimuistin koon vain 50 megatavuun [23]. Selaimen tallennusrajapintaa hyödyntämällä voidaan tarkistaa tallennustilan suuruus listauksen 2.2 mukaisesti.

Tallennustilan koko	**Sivuston raja**	**Kokonaisraja**
$<$ 8GB	20%	50MB
$>$ 8 - 32GB	20%	500MB
$>$ 32 - 128GB	20%	4% Levytilasta
$>$ 128GB	20%	4% tai 20GB (kumpi on pienempi)

Taulukko 2.1: Service Workerin tallennustilan koko massamuistista riippuen [23].

```
if ('storage' in navigator &&
    'estimate' in navigator.storage) {
    navigator.storage.estimate()
      .then(function(estimate){
        console.log(`Using ${estimate.usage} out
        of ${estimate.quota} bytes.`);
      });
}
```

Listaus 2.2: Selaimen tallennustilan koko tavuina.

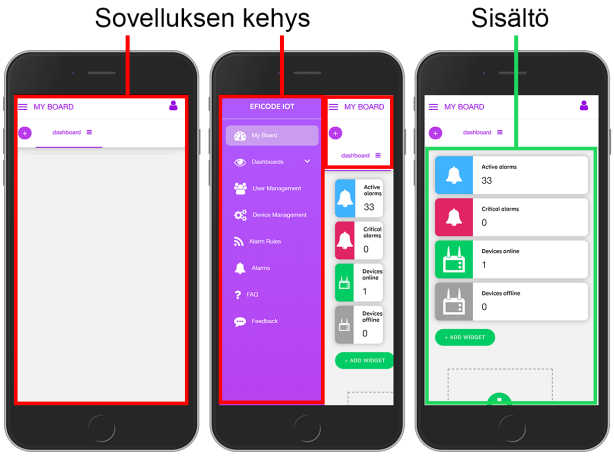
HTTPS-liikenne tarvitaan, jotta sovellus voidaan laskea PWA-sovellukseksi. Kaikki PWA-sovelluksen liikenne tulee aina lähettää salattuna HTTPS:n (Hypertext Transfer Protocol Secure) ylitse. Salatun liikenteen käyttäminen on nykyisin hyvä tapa, jotta sovellus voidaan laskea luotettavaksi. Vuodesta 2018 eteenpäin Google on myös arvioinut verkkosivuja sen perusteella käyttävätkö ne HTTPS-protokollaa [24]. Ilman salatun yhteyden käyttämistä sovellusta ei lasketa PWA-sovellukseksi ja sen arviointi on Googlen hakutuloksissa huonompi kuin salattua liikennettä käyttävien sivustojen. Tietoturvan takia Service Workeria ei pysty rekisteröimään mikäli liikenne ei ole salattu [3]. Service Workerilla voitaisiin salaamattomassa liikenteessä esimerkiksi kaapata verkkokutsuja.

Salatun liikenteen lisäksi PWA-sovellus hyödyntää niin sanottua "turvallisuus ensin-politiikkaa. Kun PWA-sovellus käynnistetään, sillä ei ole mitään oikeuksia käyttäjän laitteeseen [25]. Tarvittavat oikeudet tulevat käyttöön sitä mukaa, kun niitä tarvitaan ja käyttäjä antaa sovellukselle tarvittavat luvat. Esimerkiksi siirryttäessä GPS:ää käyttävään kohtaan sovelluksessa tulee käyttäjälle pyyntö päästä GPS-anturiin käsiksi. Mikäli käyttäjä sallii tämän, saadaan sovellukselle käyttöokeuksia lisää. Sivusto siis muodostuu vähitellen täysiveriseksi sovellukseksi, kun se saa käyttöoikeuksia lisää selaimen tukemien PWA-ominaisuuksien rajoissa [26].

2.4 Sovelluskehys

Yksi PWA-sovelluksen keskeisimmistä arkkitehtuurillisista suunnittelumalleista on, että tietyt resurssit, kuten esimerkiksi navigaatiopalkki on ladattu niin sanotuksi kehykseksi. [27] Sovelluksella on siis kehys, jossa voidaan näyttää esimerkiksi sovelluksen navigaatio ja kehyksen sisään ladataan dynaamisia näkymiä Service Workerin avulla. Kehys avautuu aina, myös laitteen ollessa verkkoyhteydettömässä tilassa. Kehys on yksi PWA-sovelluksen lähtökohdista, mutta se ei ole pakollinen vaatimus. Tarkoituksena on saada ladattua sovelluksen käyttöliittymä lähes välittömästi. Etuna kehyksessä on se, että sitä ei tarvitse aina ladata uudestaan mikäli käyttäjä vierailee sivustolla usein. Dynaamista sisältöä voidaan ladata jälkikäteen ja myös taustalla. PWA-sovelluksen tulisi aina näyttää käyttäjälleen jotakin [26]. Esimerkiksi verkkoyhteyden puuttuessa voitaisiin kehyksessä näyttää virheviesti verkkoyhteyden puuttumisesta, tai vanhaa sisältöä välimuistista. Sovelluksen latautuessa näytetään latausikkuna- tai animaatio. Sovelluskehys määrityksen mukaan navigaatio ja sisältönäkymä ovat erillisiä näkymiä. [28].

PWA-sovelluksen on tarkoitus lyhentää latausaikaa jokaisella kerralla, kun käyttäjä vierailee sivustolla. Behlin ja Rajn työssä [27] oli otettu mittaukset viidestä eri PWA-sovelluksesta ja latausajat oli otettu talteen jokaiselta kerralta. Työssä esiteltiin kuva, josta nähtiin, että latausaika peräkkäisillä vierailuilla tippui jokaisella kerralla vähän. Riippuen verkkoyhteydestä ja palvelimesta, latausaika saattoi hieman vaihdella. Kehyksen tarkoitus on siis tarjota käyttöliittymä lähes välittömästi käyttäjän laitteen välimuistista, ja vain data on tarkoitus hakea dynaamisesti palvelimelta. Kehysmalli tekee sovelluksesta nopean ja luotettavan, kun valikoita ei tarvitse joka kerta ladata palvelimelta uudestaan. Verkkoyhteydettömässä tilassa valikot voidaan näyttää välimuistista, ja kehyksen sisällä voidaan esittää esimerkiksi virheviesti.

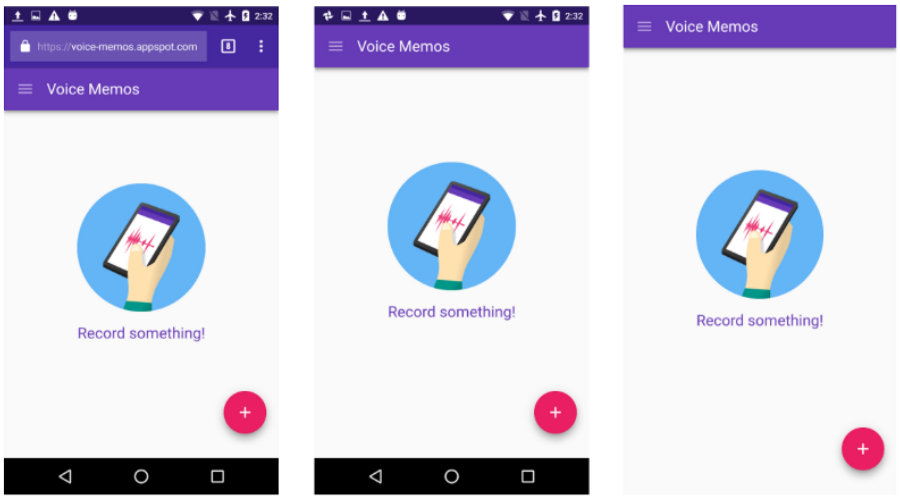


Kuva 2.4: Service Workerin välimuistiin tallennettu kehys latautuu välittömästi toistuvilla latauksilla ja näkymään voidaan ladata dynaamisesti sisältöä käytön aikana.

2.5 Näyttötilat

PWA tuo mukanaan niin sanotut näyttötilat, joiden tarkoitus on piilottaa käyttäjältä ylimääräiset valikot ja luoda sovelluksesta näin natiivisovelluksen kaltainen. Näyttötilojen esimerkit ovat kuvassa 2.5. Näyttötiloja on neljä erilaista [29]:

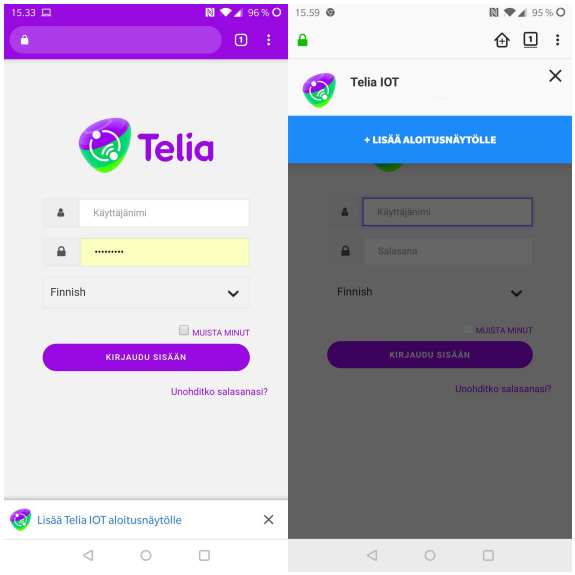
- Koko näytön tila (fullscreen) on tila jolloin laitteen koko näyttö on käytössä ja selaimen user agent on piilotettuna.
- Erillistilassa (standalone) sovellus näyttää ja tuntuu omalta sovellukseltaan. Sovellusta voidaan ajaa omassa ikkunassaan ja sillä voi olla oma ikoni käynnistysvalikossa. Selaimen käyttöliittymä on piilotettuna, mutta mobiililaitteen omat valikot ovat esillä, kuten tilapalkki.
- Pienin mahdollinen käyttöliittymätila (minimal-ui) on sama kuin erillistila, mutta mukana on vain minimaalinen määrä käyttöliittymäelementtejä navigointia varten. Käyttöliittymäelementtien määrä riippuu laitteella käytettävästä selaimesta.
- Selaintila (browser) on oletustila, jolloin sovellus aukeaa selaimen välilehdellä, tai uudessa ikkunassa, riippuen selaimesta ja laitealustasta.



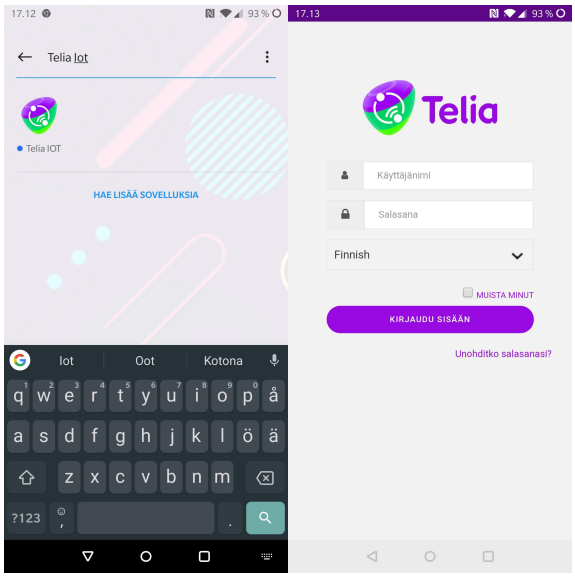
Kuva 2.5: Näyttötilat vasemmalta oikealle: Browser, standalone / minimal-ui ja fullscreen.

Kun kaikki PWA-sovelluksen ehdot on täytetty, tulee käyttäjän ruudulle kuvan 2.6 kaltainen ponnahdusikkuna.

Kun PWA-sovellus on asennettu mobiililaitteelle, tulee sovellusvalikkoon sovelluksen pikakuvake josta sovellus avautuu manifestissä määritellyllä näyttötilalla. Esimerkki kuvassa 2.7.



Kuva 2.6: Telia IoT PWA-sovelluksen ponnahdusikkuna Google-Chrome ja Firefox mobiiliselaimissa.

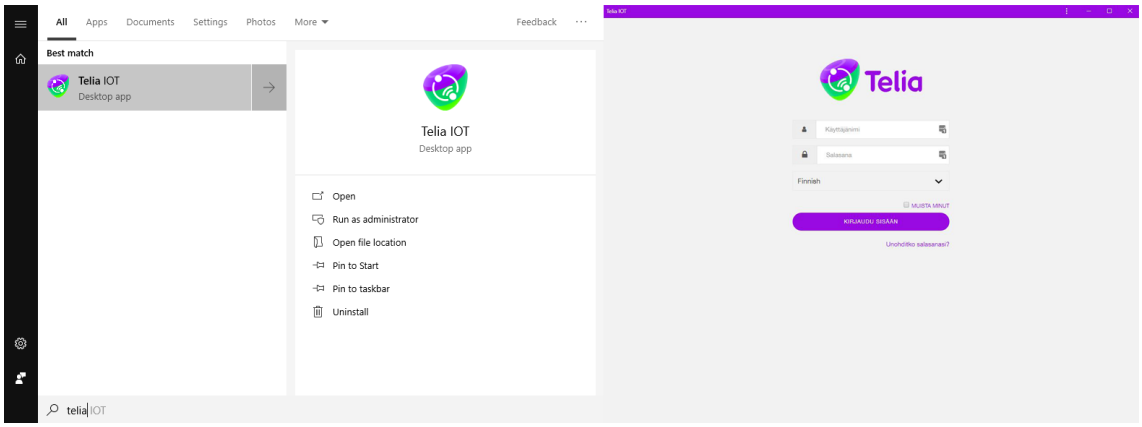


Kuva 2.7: Vasemmalla Telia IoT -sovellus asennettuna Android puhelimen sovellusvalikossa ja oikealla sovellus avattuna kokonäytön tilassa.

2.6 PWA-työasemasovellus

Chrome-selaimen versiosta 70 eteenpäin PWA-sovelluksia on mahdollista asentaa myös työasemille. PWA-työasemasovellus voidaan asentaa mille tahansa työasemalle (Linux, Windows, Mac, Chrome OS). Sovellusta voidaan käyttää käyttäjän koneella kuten muitakin työasemasovelluksia. Sovellukset tuntuvat integroiduilta ja nopeilta, sillä ne avataan käyttöjärjestelmän sovellusvalikosta, kuten mikä tahansa muukin sovellus. Esimerkki asennetusta PWA-sovelluksesta Windows-työasemalla kuvassa 2.8. Sovellus avautuu käyttöjärjestelmän natiivi-ikkunassa. Service Workerin välimuisti toimii sekä työasemalla että mobiililaitteilla, joten PWA-sovelluksen käyttäminen on luotettavaa ja nopeaa. PWA-teknologialla saadaan asennettava sovellus mobiililaitteille ja työasemille.

Google Developer -portaalin työasema-artikkelin mukaan [15] mobiililaitteiden käyttö ajoittuu yleensä iltoihin ja aamuihin, kun ihmiset ovat vapaalla. Päivällä ihmisten ollessa töissä käytetään yleensä enimmäkseen työasemasovelluksia. Googlen mukaan työasemalle asennettu sovellus tuntuu käyttäjän mielestä luotettavammalta ja nopeammalta, mikäli se löytyy suoraan työaseman sovellusvalikosta. Työasema sovelluksia ohjelmoitaessa tulee ottaa huomioon samat vaatimukset kuin mobiililaitteilla. Myös sovelluksen responsiiviseen suunnitteluun tulee kiinnittää huomiota, sillä käyttäjä saattaa venyttää työasemalla sovelluksen mihin tahansa kokoon.



Kuva 2.8: Kuvakaappaus Telia IoT -sovelluksen PWA-asennuksesta Windows-työasemalla.

3 PWA-sovellusten ominaispiirteet

Tässä luvussa vertaillaan PWA-sovelluksia yleisesti muihin teknologioihin. Muita mobiiliteknologioita ovat hybridisovellukset, React Native sovellukset ja natiivisovellukset. PWA-sovellusten mittaamista ja nopeutta tarkastellaan Googlen tapaustutkimuksen kautta.

3.1 Vertailua muihin teknologioihin

Mobiililaitteiden kenttä muuttui täysin Applen julkaistua iPhonen [30]. Applen alkuperäinen idea oli käyttää nimenomaan web-teknologioita [30]. Kuitenkin kolme vuotta julkaisun jälkeen natiivisovellukset olivat ottaneet vallan, yleensä suorituskyvyn takia. Tästä seurasi aiemmin mainittu ongelma; sovelluksia tuli kehittää erikseen jokaiselle laitealustalle [30]. Sovelluksen kehittäminen erikseen jokaiselle laitteelle on kalliimpaa ja pienemmillä yrityksillä ei välttämättä ollut rahaa tukea kuin vain toista laitealustaa, iOS- tai Android-alustaa. Lisäksi ohjelmistojen ylläpito usealle laitteelle on kallista.

3D-sovelluksissa ja peleissä natiivikielen käyttö saattaa tuoda etuja, mutta hyvin rakennetuissa verkkosovelluksissa suorituskykyeroa on miltei mahdoton huomata [30]. Sovelluskehitystä hankaloittaa lisäksi se, että jokaisella laitteella on oltava oma kehitysympäristö ja -työkalut. Rajapinnat ovat erilaiset ja sovelluksen paketointi ei ole yhdenmukaista. Kun sovelluksen ylläpitoon ja kehitykseen täytyy käyttää useampia ympäristöjä, se nostaa sovelluskehityksen kuluja [6]. Loppukäyttäjän näkökulmasta natiivisovellus tarjoaa kuitenkin yleensä parhaan käyttökokemuksen.

Yhteistä kaikille natiivisti tai natiivin kaltaisesti julkaistaville sovelluksille: natiivisovellus, hybridisovellus, tai React Native sovellus on se, että ne julkaistaan ja jaellaan jonkun sovelluskaupan kautta [6]. iOS-käyttöjärjestelmällä Apple App Store, Androidilla Google Play, Microsoftilla Windows Phone Store ja BlackBerryllä App World. Jokainen laitevalmistaja omistaa oman sovelluskauppansa. Monopoliasemassa ovat kuitenkin Android 69,6 % asennuksien määrässä ja toisena iOS 20.9 % asennusmäärällä [6]. Sovelluksia kehitettäessä ja julkaistaessa tulee noudattaa käyttöjärjestelmävalmistajien ehtoja ja ohjelmointimalleja. Tästä syystä on muodostunut ilmiö nimeltään pirstoutuminen, sillä sovelluksia tulee kehittää ja ylläpitää usealle alustalle. Uusia toimintoja voi olla vaikeaa kehittää ja julkaista samaan aikaan eri laitteille, jos kyseessä on natiivisovellus.

Hybridisovellukset tulivat paikkaamaan tätä pirstoutumista. Hybridisovellusten idea on yksinkertaistaa kehitystyötä ja ylläpitoa, sekä säästää aikaa ja rahaa. Hybridisovelluksen tarkoituksena on tarjota lähes natiivisovelluksen kaltainen käyttökokemus ja toimia mahdollisimman monella laitteella. Hybridisovellukset siis kirjoitetaan kerran ja käännetään sen jälkeen mahdollisimman monelle laitteelle.

3.1.1 Hybridisovellukset

Eri laitealustoista huolimatta kaikissa laitteissa on verkkoselain. Vuonna 2008 iPhoneDevCamp -tapahtumassa Eric Oesterle, Rob Ellis ja Brock Whitten [30] esittelivät uuden teknologian. Jokaisella alustalla oli mahdollista käynnistää selain niin sanottuun "Chromeless"-tilaan ja kutsua laitteen natiivirajapintaa JavaScriptin välityksellä. Sovellus voitiin kehittää yleisimmillä web-tekniikoilla ja julkaista mobiilisovelluksena. Tästä muodostui myöhemmin työkalu nimeltään PhoneGap. Työkalu julkaistiin myöhemmin myös muille alustoille.

Adobe lahjoitti PhoneGap-työkalun Apache Software Foundationille. Työkalun nimeksi tuli Cordova [31]. Cordovan toimintaa voidaan laajentaa erilaisilla lisäosilla "plugins", joita löytyy esimerkiksi Android-alustalle yli 4000 kappaletta [32]. Hybridisovellukset eivät siis ole natiivisovelluksia, vaan ne toimivat laitteen Webview-näkymässä web-sovelluksena [33]. Mobiililaitteissa, joissa Webview on vanhentunut, ei Cordovalla saada täysiä hyötyjä [33]. Ongelmia voi muodostua esimerkiksi suorituskyvyn tai puuttuvien funktioiden muodossa. Hybridisovelluskehitystä tutkivassa artikkelissa Apache Cordovalla käännetty sovellus käytti noin 20 % enemmän prosessoritehoa kuin natiivisovellus [33]. Hybridisovellusten etuna on kuitenkin yksi yhtenäinen koodipohja. Ne soveltuvat hyvin yksinkertaisiin sovelluksiin, joissa ei ole raskaita animaatioita tai siirtymiä näkymien välillä.

Vuonna 2013 Intel alkoi kehittää Crosswalk-nimistä projektia [34]. Crosswalk-projektin tarkoituksena oli tuoda uusimmat web-selainten ominaisuudet Cordovaan [33], sillä sovelluksia ajettiin ennen webview-näkymässä, jossa ei ollut selainten uusimpia rajapintoja. Vuonna 2017 projekti lopetettiin, sillä Android-käyttöjärjestelmien webview-näkymä päivitettiin jakamaan koodia Chrome-selaimen kanssa. Tämä mahdollisti tuoreimpien rajapintojen käyttämisen hybridisovelluksissa ilman Crosswalk-lisäosaa. Nykyisin Cordovalla käännetyt projektit tukevat siis suoraan uusimpia selainten ominaisuuksia. Hyvin tehtyä hybridisovellusta voi olla vaikea erottaa natiivisovelluksesta. Koska hybridisovellus on paketoitu natiivisovellukseksi, tapahtuu niiden jakelu natiivisovellusten tapaan. Hybridisovellusten etuna on kehitysnopeus, koska samaa koodia voidaan käyttää jokaiselle laitteelle. Ainoastaan lisäosia käytettäessä tulee tarkistaa, että jokaiselle laitealustalle löytyy oikea laajennus.

Hybridisovellukset ovat siis web-pohjaisia HTML- ja JavaScript-sovelluksia, jolloin laitealustan natiivia ohjelmointikieltä ei välttämättä tarvita ollenkaan. Hybridisovellukset upottavat sovelluksen mobiililaitteen natiivin kehyksen sisään. Kuten verkkosovelluksissa, myös hybridisovelluksissa koodi suoritetaan selaimessa. Erona verkkosovellukseen hybridisovellus paketoidaan ja ladataan sovelluskauppaan.

3.1.2 React Native

React Native on Facebookin kehittämä natiivisovelluskehys. React Nativella tehdään natiiveja sovelluksia, joiden käyttöliittymä kehitetään JavaScriptillä ja Reactilla [35]. React Nativella on mahdollista kirjoittaa koko sovellus käyttäen Reactia ja JavaScriptiä, tai vain osa sovelluksesta ja kirjoittaa loput natiivikoodilla. Esimer-

kiksi Facebook-sovellus toimii React Nativella. Muita React Nativella julkaistuja sovelluksia ovat esimerkiksi Instagram, Uber ja Skype. Tämä kertoo siitä, että teknologia on todettu hyvin toimivaksi ja se soveltuu myös vaativiin projekteihin, joissa ei ole varaa heikkoon suorituskykyyn.

React Native eroaa hybridisovelluksista ja web-sovelluksista siten, että sovelluksen näkymän renderöimisessä React Native kutsuu joko Objective C -rajapintaa iOS-laitteilla, tai Java-rajapintaa Android-laitteilla [36]. Tämä erottaa React Nativen hybridisovelluksista ja PWA-sovelluksista, jotka yleensä renderöivät selainpohjaisen näkymän. React Native voisi tukea Android- ja iOS-alustan lisäksi myös muita käyttöjärjestelmiä, teoriassa esimerkiksi älytelevisioita, mutta silloin jonkun täytyisi ohjelmoida puuttuva rajapinta.

React Nativen etuna on natiivisovelluksen nopeus sekä yksi ja sama koodipohja molemmille käyttöjärjestelmille. Sovelluksen kehittämiseen kuluu huomattavasti vähemmän aikaa ja rahaa. Sovellus voidaan luoda vain käyttäen yhtä ohjelmointikieltä, eikä ole tarvetta etsiä erikseen iOS tai Android kehittäjää. React Native on ohjelmistokehys, jota voi itse laajentaa kirjoittamalla natiivikoodia. Mikäli kehys ei suoraan tue jotain toimintoa, voidaan sellainen kirjoittaa niiden rajoitusten puitteissa kuin iOS- ja Android-alustoilla on.

3.1.3 Natiivisovellus

Android- ja iOS-alustoja kehitetään eri ohjelmointikielillä, siksi molemmille alustoille tarvitaan: kehittäjät jotka tuntevat laitealustan, omat kehitysympäristöt ja kääntäjät. Mikäli mobiilisovelluksen haluaa kehittää natiivisovelluksena molemmille laitealustoille, saattavat kehityskustannukset kasvaa suuremmiksi kuin hybridi-, tai PWA-sovelluksissa siksi, että yksi ja sama lähdekoodi ei kelpaa molemmille laitteille.

Sovelluskehityksen näkökulmasta molemmat laitealustat ovat hyvin vakaita ja ne ovat olleet käytössä useita vuosia. Natiivisovellus toimii nimensä mukaisesti natiivilla nopeudella [37]. Hybridisovelluksia taas suoritetaan ylimääräisen ympäristön sisällä, joka vie laitteesta suoritintehoa ja saattaa hidastaa sovelluksen toimintaa.

Kahden eri koodipohjan ylläpito voi olla myös haastavaa. Toinen sovelluksista voi esimerkiksi olla edellä toista päivityksissä, koska sovelluksia kehitetään eri ohjelmointikielillä ja julkaistaan eri sovelluskaupoissa.

Myös käyttöliittymäkirjastot ovat erilaisia. Esimerkiksi iOS-laitteilla on vain yksi kotinäppäin. Android-laitteilla sen sijaan on kolme kotinäppäintä. [38] Käyttöliittymä tulee suunnitella kummallekin laitteelle sopivaksi. iOS-laitteella esimerkiksi navigointi taaksepäin tulee hoitaa sovelluksesta, kun taas Android-laitteella sovelluksessa voidaan navigoida taaksepäin natiiveista kotinäppäimistä. iOS-laitteella navigaatiopalkki on keskitetty, kun taas Android-laitteella navigaatiopalkki alkaa reunasta. Lisäksi tekstityypit, listat, ikonit ja taulukot ovat kummallakin laitteella erilaisia.

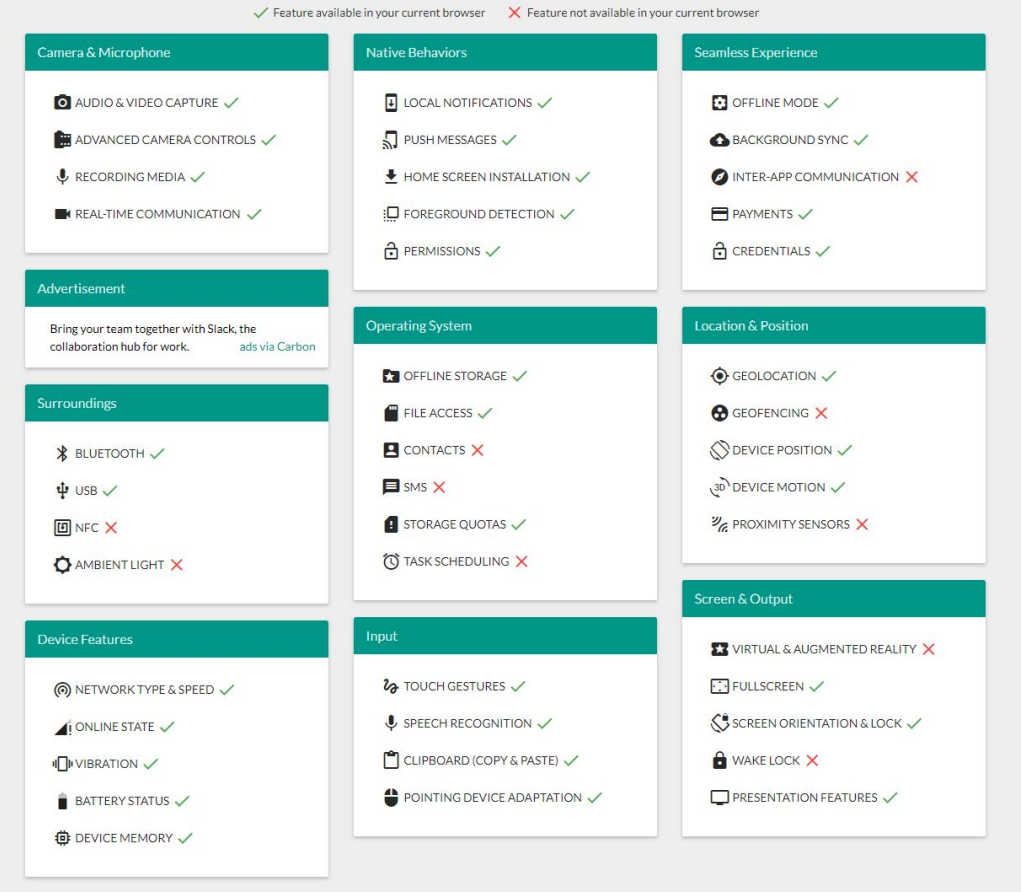
Natiivisovellus on järkevä teknologia, jos tarkoituksena on käyttää paljon laitteen natiiveja rajapintoja. Natiivisovelluskehitys soveltuu hyvin peleihin parhaan suori-

tuskyvyn takia.

3.2 Natiivi- ja hybridisovellukset verrattuna PWA-sovelluksiin

PWA-sovellus on muista sovellustyypeistä erilainen, sillä sitä ei paketoida eikä jaeta sovelluskaupoissa. Sovellus voidaan löytää suoraan verkosta normaalin verkkosivun tapaan. Etuna on se, että sovellusta voidaan käyttää heti tekemättä sovelluksen asennuspäätöstä, luovuttamatta käyttöoikeuksia tai hyväksymällä sovelluksen käyttöehtoja. PWA-sovellus on siis verkkosovellus, joka voidaan asentaa käyttäjän laitteelle pikakuvakkeeksi.

PWA-sovellusten suurin ongelma toistaiseksi on sovelluksen pääsy laitteiden natiiveihin rajapintoihin, vaikka verkkoselaimen rajapinnat ovatkin jo kattavat. Selaimen tukemia rajapintoja on esitelty kuvassa 3.1. Mikäli sovelluksessa on tarkoitus käyttää jotain osa-aluetta jota selain ei tue, silloin on luonnollista tehdä natiivisovellus.



Kuva 3.1: Google Chrome Version 72 tukemat rajapinnat. Kuvakaappaus sivulta https://whatwebcando.today/

Koska PWA-sovellus on vain pikakuvake, on sovelluksen asennuskoko pieni. PWA-sovellus ei vie juurikaan tallennustilaa käyttäjän laitteelta, sillä kaikki data haetaan verkosta. Ainoastaan välimuistiin tallennettavat asiat vievät tilaa käyttäjän laitteelta.

PWA-sovellus on aina ajan tasalla, sillä käyttäjän laitteelle on asennettu ainoastaan pikakuvake ja varsinaisen sovelluksen tiedot käydään hakemassa aina palvelimelta, ellei tietoa ole tallennettu välimuistiin. PWA-sovellus on myös mahdollista kääntää hybridisovellukseksi, mikäli sen haluaa julkaista sovelluskaupassa.

PWA- ja hybridisovellukset alkavat olla melko lähellä toisiaan. Hybridisovellukset toteuttavat Android-laitteella WebView järjestelmäkomponentin, joka mahdollistaa sovelluksen näyttää tietoa verkosta sovelluksen sisällä. Android-laitteissa on kaksi tapaa näyttää verkkosisältöä. Perinteisen verkkoselaimen kautta, tai WebView-näkymän kautta sovelluksen sisällä. Androidin versiosta 4.4 [39] lähtien WebView-komponentti on perustunut Chromium-selaimeen. Tämä mahdollistaa uusimpien verkkotekniikoiden käyttämisen ilman että itse Android-järjestelmää päivitettäisiin. Aikaisemmin kehittäjät ovat joutuneet odottamaan WebView-komponentin päivittymistä tai keksimään muita ratkaisuja toimintojen suorittamiseksi. Hybridisovellukset eroavat PWA-sovelluksista käyttämällä natiivirajapintoja sovelluksen ja laitteen välillä. Hybridisovellus paketoidaan tietylle laitealustalle ja julkaistaan laittevalmistajan jakelukanavalla. PWA-sovelluksen etuna on se, että kehittäjien ei tarvitse noudattaa minkään sovelluskaupan käyttöehtoja. Sovelluksen julkaisusta ei myöskään tarvitse maksaa mitään. PWA-sovelluksen etuna voi myös olla löydettävyys. Ihmiset etsivät useimmiten tietoa verkon hakukoneista. Jotta yksittäinen käyttäjä etsisi sovellusta sovelluskaupasta, täytyy sovelluksen olla tuttu muuta kautta.

Suuri etu PWA-sovelluksissa on niiden levytilavaatimus. Pinterestin tekemässä tapaustutkimuksessa [14] Pinterestin sovelluskaupoissa julkaiseman sovelluksen koko oli Android-alustalla noin 10 megatavua ja iOS-alustalla 56 megatavua. Pinterestin tekemien PWA-muutosten myötä PWA-sovelluksen koko oli 150 kilotavua. Vaikka Pinterest edelleen jakelee natiivisovellusta Android- ja iOS-alustoille, he pystyivät tekemään PWA-sovelluksella saman kuin mobiilisovelluksella, mutta vain murto-osalla sovelluksen koosta. Tämä on käyttäjälle etu, suurta natiivisovellusta ei tarvitse ladata ja asentaa, jotta sitä pääsee käyttämään. PWA-sovellusta voi käyttää verkossa heti, ilman että on ladannut yhtään mitään tai edes tehnyt päätöstä sovelluksen asentamisesta. Pinterestin suorituskykymittauksen mukaan [14] sovelluksen kääntäminen nopeutui 30 % ja parsiminen 25 % toisella vierailulla.

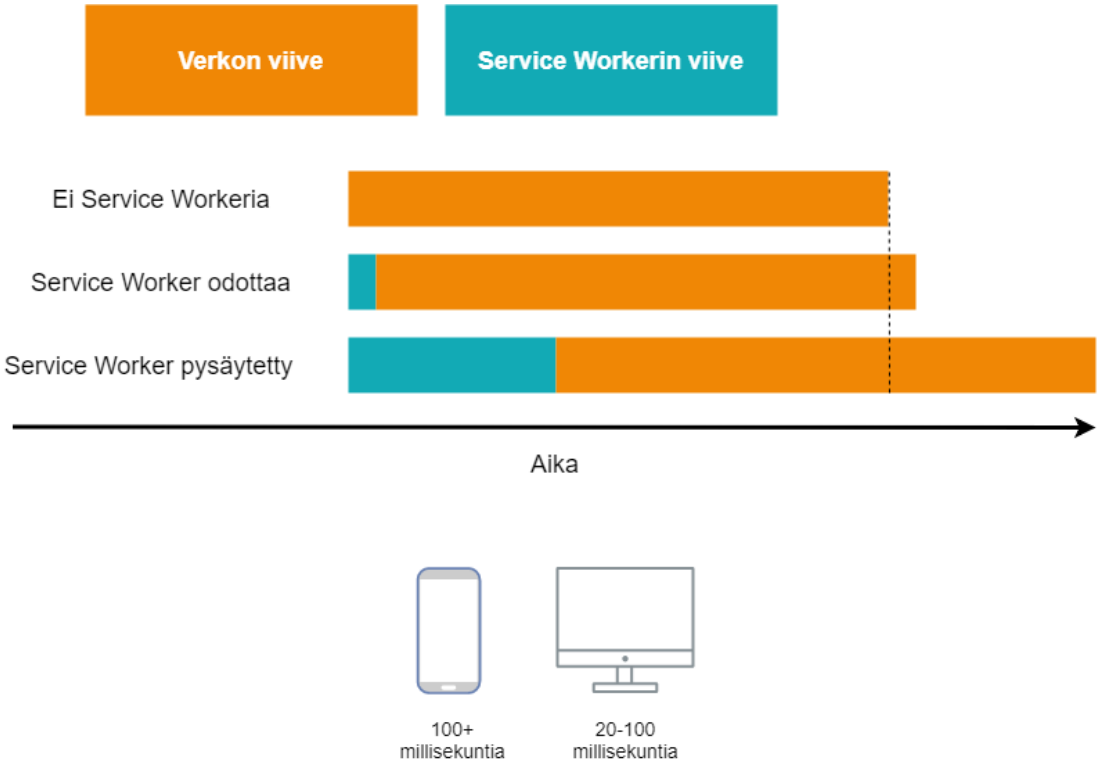
Hybridi- natiivi ja PWA-sovellusten eroja on esitelty taulukossa 3.1.

	Natiivi	**React Native**	**Hybrid**	**PWA**
Koodin uudelleen-käytettävyys	Huono	Keskiverto	Hyvä	Erinomainen
Suosio	Erinomainen	Keskiverto	Keskiverto	Keskiverto
Rajapinnat	Erinomainen	Hyvä	Keskiverto	Huono
Oppimiskäyrä	Huono	Keskiverto	Hyvä	Erinomainen
Jakelukanava	Sovelluskauppa	Sovelluskauppa	Sovelluskauppa	URL / Windows marketplace
Ohjelmointikieli	Natiivikieli, Java / Swift	JavaScript, React, tarvittaessa Natiivikieli	JavaScript, HTML, CSS, Natiivikieli laajennuksille	JavaScript, HTML, CSS
Renderöinti	Natiivi	Natiivi	Selain / Webview	Selain
Time To Market	Huono	Keskiverto	Hyvä	Erinomainen
Hintaluokka	Kallis	Kallis	Edullinen	Halvin
Testattavuus	Fyysinen Laite	Fyysinen Laite	Selain	Selain
Asennettavissa	Kyllä	Kyllä	Kyllä	Kyllä
Offline tuki	Kyllä	Kyllä	Kyllä	Kyllä
Testattavissa ennen asennusta	Ei	Ei	Ei	Kyllä
Sovellusilmoitukset	Kyllä	Kyllä	Kyllä	Kyllä
Laiterajapinta API	Kyllä	Kyllä	Kyllä	Rajoitettu
Synkronointi taustalla	Kyllä	Kyllä	Kyllä	Kyllä

Taulukko 3.1: Teknologioiden erot [3]

3.3 PWA-sovelluksen suorituskyky ja mittaaminen

Googlen developer dokumentaatiossa tapaustutkimusten osastolla Phil Waltonin artikkelissa [18] on mitattu Google Analyticsiä hyödyntäen Googlen omaa I/O websovelluksen nopeutta tuotannossa oikeilla käyttäjillä. Hyödyntämällä Google Analytics -työkalua yhdessä Service Workerin kanssa saadaan vastaus, miten Service Workerin käyttö näkyy loppukäyttäjälle. Pelkästään Lighthouse-työkalua hyödyntämällä, tai ajamalla kehittäjän koneella testejä, ei saada tarpeeksi kattavia vastauksia. Testien tuloksiin vaikuttaa esimerkiksi se, onko käyttäjän Service Worker aktiivinen selaimessa. Mikäli selain on tallettanut useita Service Workereita se sammuttaa niitä tarpeen mukaan. Kun käyttäjä palaa takaisin sivulle, jonka Service Worker on jo olemassa, menee sen käynnistymisessä pieni hetki, joka taas hetkellisesti nostaa latausaikaa. Chrome Dev Summit-tapahtumassa oli mitattu Service Workerin käynnistysaikaa sekä tietokoneella että mobiililaitteella [20]. Mittausten tuloksia on esitelty kuvassa 3.2.



Kuva 3.2: Service Workerin käynnistysaika eri tilanteissa, sekä käynnistysaika keskimäärin mobiililaitteella ja työasemalla.

Konferenssissä esitettyjen lukujen perusteella Service Workerin käynnistysaika on tietokoneella keskimäärin 20-100 millisekuntia ja mobiililaitteella enemmän kuin 100 millisekuntia. Service Worker ei siis ole aina aktiivinen, mikä hidastaa sen käyttöä. Suorituskyvyn kannalta Service Workerin käynnistysaika ei ole ilmaista. Välimuistiluku ei ole aina välitöntä. Agressiivinen välimuistin käyttö Service Workerissa voi

viedä kaistaa pääsivulta. Jos esimerkiksi kaikki sivuston kuvat laitetaan välimuistiin, ne voidaan ladata ennen kuin sovelluksen tärkeämmät tiedot on haettu verkosta.

Tarkastellaan Googlen tapaustutkimuksen kautta mittaustuloksia, miten sovelluksen käyttö eroaa loppukäyttäjälle Service Workerin ollessa käytössä.

1. Onko Service Workerin välimuisti parempi kuin selaimessa jo oleva HTTP välimuisti?

Oletus on, että sivustolle uudelleenpalaava käyttäjä saa sivun ladattua nopeammin kuin ensimmäistä kertaa sivustolla vieraileva käyttäjä, koska selain tallettaa joitain pyyntöjä välimuistiin oletuksena. Service Workerin avulla kehittäjä voi valita, mitä tiedostoja sovelluksesta talletetaan. Tutkimuksesta käy ilmi, että Service Workerin välimuistin käyttö on nopeampaa kaikissa eri lähtötilanteissa. Vaikka selaimen välimuistilla päästään ensimmäisessä latauksessa hyvin lähelle samaa lopputulosta on Service Worker silti jonkin verran nopeampi.

2. Kuinka paljon Service Workerin käyttö vaikuttaa sivun käytettävyyteen ja latausaikaan?

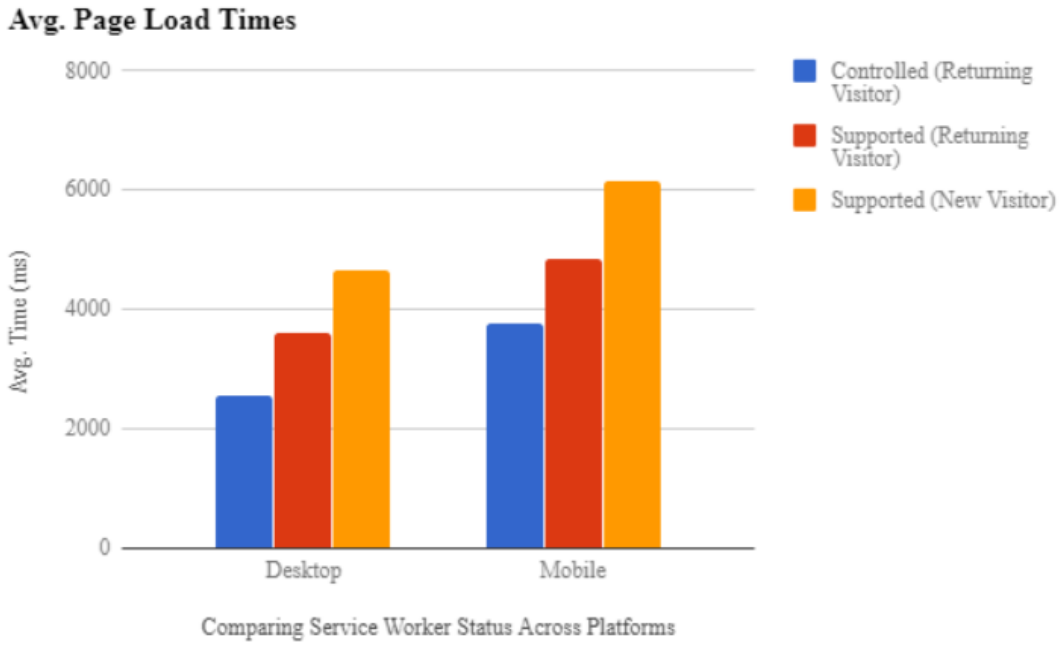
Kuinka nopealta sivu tuntuu käyttäjälle, riippumatta oikeasta latausajasta ja perinteisistä latausmetriikoista? Kysymykseen ei ole helppoa vastata ja Walton [18] sanookin, että mikään metriikka ei ole täydellinen näin subjektiiviseen kysymykseen. Osa mittauksista on kuitenkin järkevämpiä kuin toiset ja tärkeintä onkin valita oikeat mittapisteet. Lisäksi Walton suositteli aina käyttämään oikeaa sovellusta oikeilla käyttäjillä suorituskyvyn arvioinnissa esimerkiksi hyödyntämällä Google Analytics-työkalua yhdessä omien mittausten kanssa. Waltonin mielestä paikallisesti kehittäjän koneella ajetut testit eivät ole tarpeeksi kattavia, sillä paikallisesti ajetut testit eivät ota esimerkiksi huomioon Service Workerin sammumista.

Waltonin tapaustutkimuksessa [18] mitattiin sivuston latausaikaa. Latausaika on hyvä mittaus ensimmäiseen kysymykseen. Se ei kuitenkaan vastaa toiseen kysymykseen. Huomioidaan vielä, että sivuston latausaika ei ole sama asia kuin se, milloin käyttäjä voi alkaa hyödyntää sovellusta. Kaksi sovellusta täysin samalla latausajalla voi tuntua täysin erilaiselta. Esimerkiksi sovellus, joka näyttää aloitusruudun tai latausindikaattorin, voi tuntua käyttäjästä huomattavasti nopeammalta kuin sovellus, joka näyttää vain tyhjää sivua muutaman sekunnin. Waltonin esimerkissä [18] metriikaksi valittiin kulunut aika ensimmäiselle piirrolle. Tämä mittaus on tärkeä, sillä se määrittelee ensikokemuksen käyttäjälle. Hyvä ensikokemus voi vaikuttaa positiivisesti sovelluksen loppukäyttöön.

Latausajan mittaukset jaettiin tapaustutkimuksessa kolmeen eri kategoriaan.

- **Controlled:** Service Worker määrittelee välimuistiin talletettavat asiat ja sovellusta voi käyttää Offline tilassa.
- **Supported:** Service Worker on tuettuna selaimessa, mutta se ei ole vielä tallettanut mitään. Kyseessä on ensimmäistä kertaa sivustolla vieraileva käyttäjä.
- **Unsupported:** Käyttäjä, jonka selain ei tue Service Workeria ollenkaan.

Kuvasta 3.3 nähdään, että Service Workerin hallitsema välimuisti (sininen pylväs) lataa sivut nopeammin kuin käyttäjillä, joilla on selaimen välimuistissa jo tietoa (punainen pylväs). Mielenkiintoista on myös huomata, että mobiilikäyttäjillä latausaika on Service Workerin ollessa käytössä lyhyempi, kuin uusilla tietokonekäyttäjillä.



Kuva 3.3: Keskimääräiset latausajat Googlen IOWA sovelluksessa. Kuvakaappaus Google Developer portaalista [18].

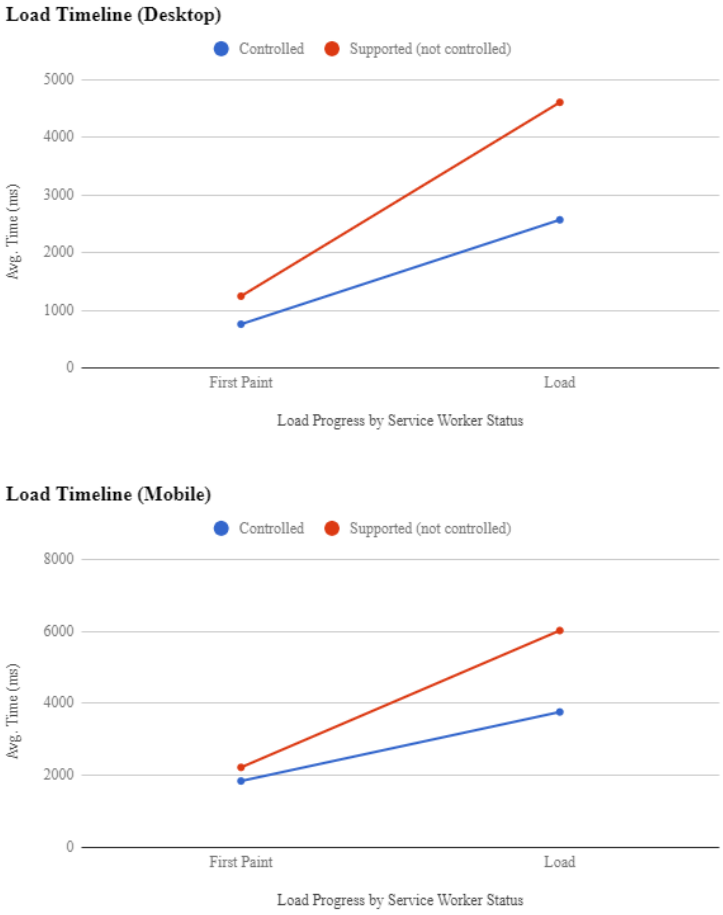
Service Workerin tila	**Käyttäjätyyppi**	**Keskimääräinen latausaika (ms)**
Käytössä	Palaava käyttäjä	2568
Tuettu, ei käytössä	Palaava käyttäjä	3612
Tuettu, ei käytössä	Uusi käyttäjä	4664

Taulukko 3.2: Keskimääräinen sivuston lataamisaika tietokoneella IOWA tapaustutkimuksessa [18].

Service Workerin tila	**Käyttäjätyyppi**	**Keskimääräinen latausaika (ms)**
Käytössä	Palaava käyttäjä	3760
Tuettu, ei käytössä	Palaava käyttäjä	4843
Tuettu, ei käytössä	Uusi käyttäjä	6158

Taulukko 3.3: Keskimääräinen sivuston lataamisaika mobiililaitteella IOWA tapaustutkimuksessa [18]

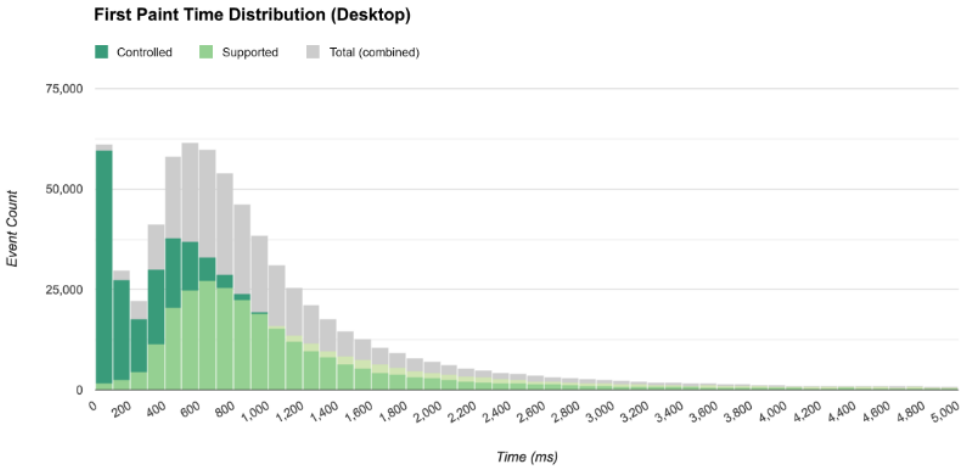
Taulukoiden 3.2 ja 3.3 latausajoista nähdään, että ero Service Workerin ollessa päällä mobiililaitteella ja verrattuna uuteen työasemavierailijaan on noin sekunnin nopeampaa mobiililaitteella. Vastatakseen toiseen kysymykseen, kuinka paljon Service Worker vaikuttaa sovelluksen käytettävyyteen, Waltonin [18] tapaustutkimuksessa otettiin mittaukset "Ensimmäinen piirto-mittapisteestä.



Kuva 3.4: Keskimääräiset ensimmäisen piirron ajat Googlen IOWA sovelluksessa. Kuvakaappaus Google Developer portaalista [18]

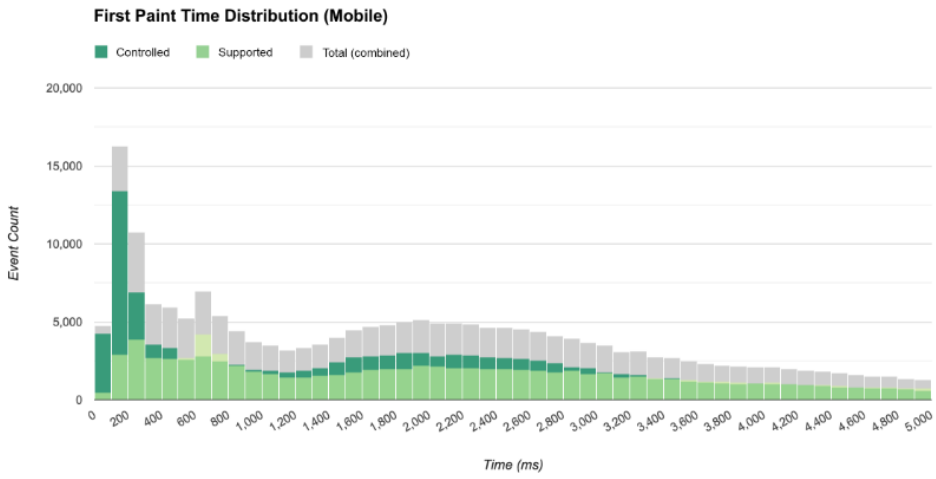
Kuvasta 3.4 käy ilmi, että Service Workerilla on paljon pienempi vaikutus "Ensimmäinen piirto-mittapisteessä mobiililaitteella. Dataa täytyy kuitenkin vielä tarkas-

tella lisää, jotta saadaan käsitys miksi asia on näin. "Ensimmäinen piirto-tuloksista on poimittu joka ikinen sivulataus ja muodostettu kuvaaja, joka esitellään kuvassa 3.5.



Kuva 3.5: Ensimmäinen piirto Service Workerilla ja ilman (Tietokone). Kuvakaappaus Google Developer portaalista [18]

Kuvasta 3.5 nähdään, että Service Workerin ollessa käytössä ensimmäinen piirtoaika on pienentynyt noin puoli sekuntia. Lisäksi Service Workerin ollessa käytössä saatiin "ensimmäinen piirto"melkein heti työasemakäytössä. Mielenkiintoista kuvan graafissa on se, että Service Workerin ollessa käytössä on jakauma kellon muotoinen, eikä laskeva. Waltonin tutkittua asiaa selvisi jakauman johtuvan siitä, että Service Worker ei ole aina aktiivinen selaimessa. Selain lopettaa Service Worker -säikeen ajamisen säästääkseen resursseja. Jokaista Service Workeria ei olisikaan järkevää pitää aktiivisena, jos olisi vieraillut sadoilla eri verkkosivuilla. Tämä selittää jakauman. Joillakin käyttäjillä Service Worker on saattanut olla sammunut ja sen uudelleen käynnistämisessä on kestänyt pieni hetki. Kuvasta nähdään myös, että vaikka Service Worker olisi sammuksissa, saadaan sen avulla ladattua sivusto nopeammin kuin selaimen tehdessä kaikki kyselyt puhtaasti verkosta.



Kuva 3.6: Ensimmäinen piirto Service Workerilla ja ilman (Mobiililaite). Kuvakaappaus Google Developer portaalista [18]

Service Workerin tila	**Tietokone**	**Mobiililaite**
Käytössä	583	1634
Tuettu, ei käytössä	912	1933

Taulukko 3.4: Mediaani ensimmäiselle piirrolle (ms)

Mobiililaitteella kuvan käyrä on melko pitkä ja eroaa huomattavasti tietokoneesta. Tämä johtuu luultavasti siitä, että mobiililaitteella Service Workerin uudelleenkäynnistys kestää luonnollisesti kauemmin kuin tietokoneella. Tämä antaa selityksen myös kuvassa 3.6 näkyvään pieneen "Ensimmäinen piirto"eroon. Tämä myös todistaa sen, että testit on järkevää suorittaa oikeilla käyttäjillä oikeassa palvelussa, jotta kaikki käyttötapaukset tulee otettua huomioon. Oikeilla käyttäjillä saadaan huomioitua tilanne, jossa Service Worker on ollut esimerkiksi sammuksissa, tai että välimuistin tilanne on ollut eri kuin lokaaleissa koetilanteissa.

Taulukosta 3.4 nähdään, mikä vaikutus Service Workerilla on sovelluksen nopeuteen. Nopeus taas vaikuttaa suoraan sovelluksen käyttökokemukseen. Niin tietokoneella kuin mobiililaitteella sovelluksen latausaika on lyhyempi, mikäli Service Worker on käytössä ja ladannut välimuistiin tietoa. Kun Service Workerista ladataan tietoa, laitteen ei tarvitse myöskään kuluttaa virtaa verkkopyyntöihin [40]. Service Workerin käyttö säästi noin 25 % dataliikenteestä ja antoi myös verkkoyhteydettömän tuen sovelluksille. Service Workerin käyttö kuluttaa kuitenkin enemmän virtaa laitteesta [17]. Vaikka verkkoliikenne pienenee, käyttää Service Worker enemmän virtaa laitteesta. Työssä tutkittiin 28 erilaista tapausta. Seitsemän eri PWA-sovellusta, kaksi eri laitetta ja kaksi eri verkkoa, Wifi ja 2G. Kahdessakymmenessä tapauksessa Service Workerin käyttö PWA-sovelluksessa lisäsi laitteen virrankulutusta. Kahdeksassa tapauksessa virrankulutus oli kuitenkin pienempää. Tutkimuksen tulos osoittaa, että kehittäjän tulee olla varovainen teknologiaa hyödyntäessä. Useimmiten virran-

käytön lisääminen on kuitenkin järkevää käyttökokemuksen kannalta. Lisäksi virrankulutuksen lisääntyminen oli hyvin pientä verrattuna sivustoon, joka ei käyttänyt Service Workeria.

4 Tutkimusongelma ja tutkimusmenetelmät

Tässä luvussa esitellään Telia-IoT palvelu, tutkimusongelma ja -menetelmät. Lähtökohtainen tutkimuskysymys muodostui Telian tarpeista saada nykyisestä toteutuksesta mobiilisovellus. Teknologiaksi valittiin kuitenkin PWA-sovellus, sillä sen kehittäminen olisi halvempaa ja nopeampaa. Tutkimuskysymykseksi muodostui millaisia etuja PWA-sovellus tuo mukanaan sovelluksen käyttäjille? Onko PWA-sovellus nopeampi kuin normaali verkkosivusto? Miten nopeuden vaikutusta tarkasteltaisiin?

4.1 Tutkimusasetelma

Telia IoT-palvelulla (Internet of Things, IoT, esineiden internet) on helppoa seurata huoneilman lämpötila-, kosteus- ja hiilidioksiditasoja. Lisäksi on mahdollista tarkastella kuinka paljon työpisteitä tai neuvotteluhuoneita käytetään. Telia IoT-palvelu on hanke, jonka sovelluskehitys on käynnistetty vuoden 2016 alussa. Sovelluksen avulla on mahdollista parantaa viihtyvyyttä ja tehokkuutta toimiston työympäristössä. Moni viettää suuren osan päivästä sisätiloissa, erityisesti toimistossa. Sisäilman laadulla onkin huomattava vaikutus työhyvinvointiin, terveyteen, työtehokkuuteen sekä viihtyvyyteen työpaikalla.

Telia IoT-palvelun avulla voi esimerkiksi seurata lämpötilaa ja sen muutoksia. Palvelun avulla voidaan tutkia, onko toimistossa esimerkiksi liian kylmä, tai nähdä nousiko lämpötila varastossa liian kuumaksi menemättä itse paikalle. Kosteutta ja hiilidioksiditasoja seuraamalla voidaan päästä jäljille sisäilman muutoksista ja seurata esimerkiksi kokoustilan ilmanlaatua. Liikesensoreilla voidaan laskea työpisteiden, tai neuvotteluhuoneiden käyttöaste.

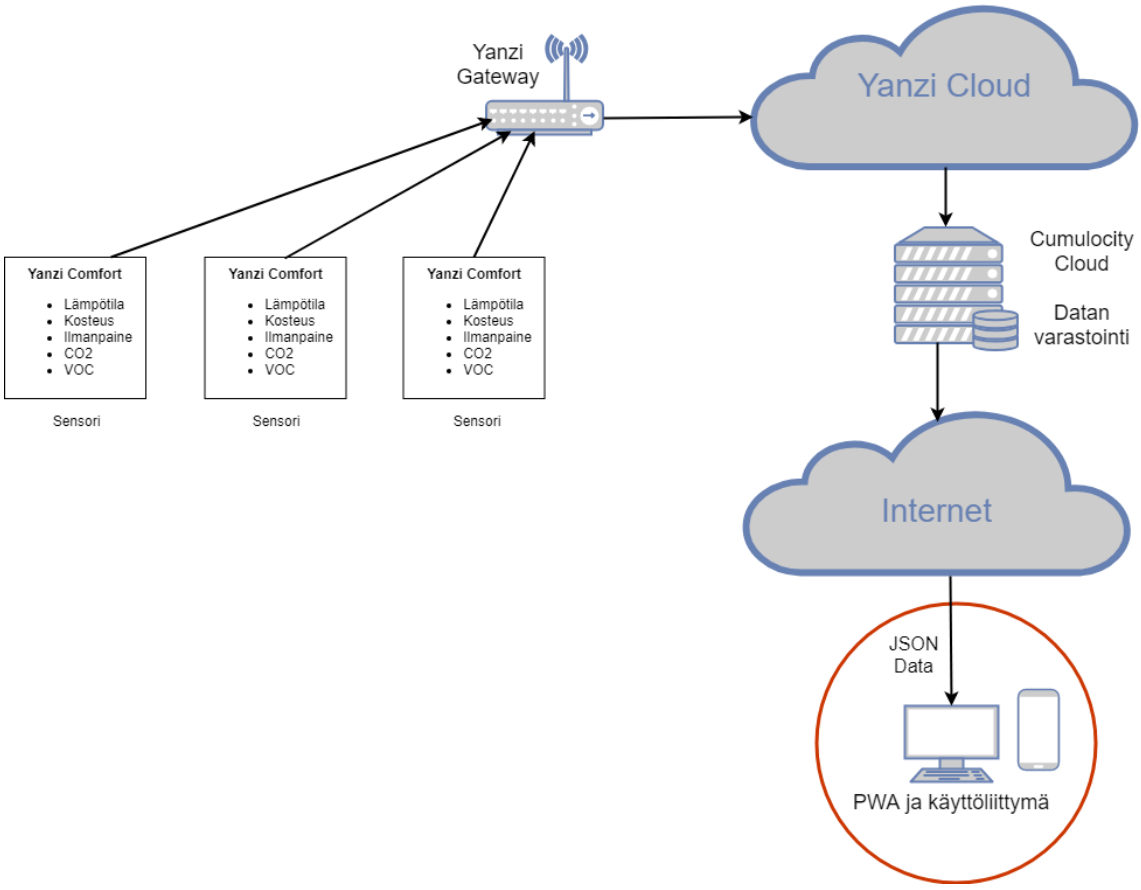
Palvelun laitteet toimittaa ruotsalainen Yanzi[1]. Telian asiakkaille tarjoamaan pakettiin kuuluu 4G-tukiasema ja lisäksi 10-12 sensoria asiakkaan valitsemasta paketista riippuen. Tukiasema käyttää 4G-verkkoa datan toimittamisessa tietovaraston palvelimelle. Tukiaseman mukana toimitetaan myös 4G SIM-kortti. Tukiasemassa on myös pieni akku, jolla laite pysyy käynnissä noin 8 tuntia hätätapausten tai sähkökatkosten aikana. Jos verkkoyhteyttä ei jostain syystä ole, on tukiasemassa pieni muisti, johon data voidaan tallentaa siihen asti, kunnes verkkoyhteys on taas saatavilla. Kun verkkoyhteys palaa, tukiasemaan lähetetään muistiin tallennettu data palvelimelle. Tiedon tallentaminen tukiasemaan väliaikaisesti mahdollistaa paljon uusia käyttökohteita ja tapoja. Palvelua on esimerkiksi testattu valtamerilaivalla, jossa verkkoyhteys on poikki useamman viikon. Data lähetetään vasta satamassa. Kaikki liikenne palvelimen ja tukiaseman välillä on salattua.

Sensorit saavat virtansa kahdesta AA-paristosta. Yanzi lupaa sensoreiden käyttöajaksi noin viisi vuotta kahdella paristolla [41]. Yhdessä sensorissa voi olla useita mittatyyppejä. Sensori voi esimerkiksi mitata lämpötilaa, kosteutta ja ilmanpainetta samanaikaisesti. Sensorialueen kantavuutta on mahdollista kasvattaa erillisillä

[1] (https://www.yanzi.se/)

toistimilla. Näin koko toimiston alue saadaan katettua. Sensori on esitelty kuvassa 4.2.

Taustapalvelut toimittaa saksalainen Cumulocity GmbH. Cumulocityn toimittamaa rajapintaa hyödyntäen IoT-käyttöliittymän data haetaan JavaScriptillä. Cumulocity toimittaa palvelun mukana myös kattavan hallintapaneelin sekä työpöytänäkymän. Sovelluksen käyttö osoittautui kuitenkin hankalaksi. Tämän seurauksena Telia käynnisti vuoden 2016 alussa projektin helposta käyttöliittymästä IoT-laitteille.



Kuva 4.1: Telia IoT-arkkitehtuurikuvaus. Tämä työ koskee punaisella ympyrällä merkittyä osaa.

IoT-sovelluksessa ei ollut tarvetta päästä käsiksi puhelimen natiiveihin rajapintoihin. Lisäksi osa PWA-sovelluksen ei pakollisista vaatimuksista oli jo täytettynä, kuten responsiiviset näkymät ja salattu yhteys. Palvelua tarjotaan asiakkaille salattuna HTTPS-protokollan ylitse, mikä on yksi PWA-sovelluksen vaatimuksista. Salatun liikenteen käyttäminen laskee sovelluksen luotettavaksi selaimessa. Sovellus on responsiivinen Bootstrap-kirjaston ansiosta, jonka avulla sovellus muotoutuu kaikille eri laitteille ja resoluutioille. Responsiivisuus ei ole pakollinen vaatimus, mutta luokitellaan yhdeksi PWA-sovelluksen vaatimukseksi.

IoT-palvelu on yksinkertaisesti verkkosivu IoT-laitteiden tuottaman datan tarkaste-

Kuva 4.2: Yanzi motion sensori, joka mittaa: liikettä, kosteutta, lämpötilaa, valoisuutta ja äänenvoimakkuutta.

luun. Sovelluksen vaatimuksena oli toimia pääpiirteittäin kaikilla mahdollisilla mobiililaitteilla, sekä vähintään uusimmilla ja yleisimmillä verkkoselaimilla.

Kehittäjätiimin kesken suoritettiin lyhyt arviointi erilaisista teknologioista ja vaihtoehdoista. Koska projektin kulut haluttiin pitää kurissa, valittiin toteutusteknologiaksi PWA-sovellus sen yksinkertaisuuden ja monimuotoisuuden vuoksi. Lisäksi projektin nykyiset kehittäjät pystyivät toteuttamaan PWA-sovelluksen samalla ohjelmointikielellä millä sovellus oli tehty. Ei myöskään ollut tarvetta palkata ja etsiä uusia kehittäjiä, joilla on kokemusta mobiilisovellukehityksestä ja iOS- tai Android-alustoista.

Kehittäjillä ei kuitenkaan ollut PWA-sovelluksista aiempaa kokemusta, eikä PWA-sovelluksia ole vielä tutkittu paljon tieteellisesti.

4.2 Tutkimuskysymykset

Sovelluksen muuttaminen PWA-sovellukseksi toi mukanaan useita kysymyksiä. Ovatko esimerkiksi vanhat projektissa käytössä olevat työkalut riittäviä PWA-sovelluksen luomiseksi? Mitä kaikkea tulisi ottaa huomioon ja mitä oli jo valmiina? Kuinka paljon nopeampi PWA-sovellus on, ja asentavatko käyttäjät sovellusta puhelimeensa? Mitä muita hyötyjä PWA-sovellus tuo? Kuinka arvioidaan PWA-sovelluksen tuomat edut ja haitat?

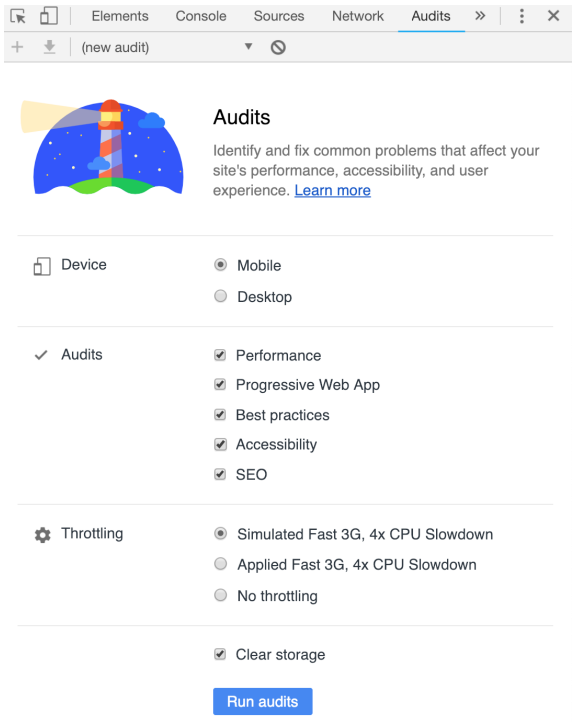
Tutkielman tutkimuskysymykset ovat seuraavat:

- **TK1:** Kuinka paljon nopeampi Service Workerin välimuisti on kuin verkkosivu, joka ei hyödynnä Service Workerin välimuistia?
- **TK2:** Miten olemassa oleva verkkosivu voidaan muokata PWA-sovellukseksi?
- **TK3:** Minkälaisia hyötyjä tai haittoja saadaan käyttämällä PWA-sovellusta?

4.3 Tutkimusmenetelmä

PWA-sovelluksen vaikutuksia käytännössä tarkastellaan Design Science -tutkimuksella. Design-tutkimus on tutkimusstrategia, joka pyrkii kehittämään sekä teoriaa että käytäntöä [42]. Design Science -tutkimusta voi käyttää niin laadulliseen kuin määrälliseen tutkimukseen. Design-tutkimus on ongelmalähtöistä oppimista, tai tutkivaa oppimista. Olemassa olevan sovelluksen muuttaminen PWA-sovellukseksi oli teoreettinen tehtävä, josta muodostui tutkimuskysymys.

Lighthouse on avoimen lähdekoodin työkalu [43], jonka pääasiallinen tarkoitus on parantaa verkkosivujen laatua antamalla pisteytyksen erilaisista kategorioista. Kategorioita ovat: suorituskyky, PWA, hakukoneoptimointi, verkon parhaat käytännöt ja saavutettavuus. Jokaisesta kategoriasta annetaan pisteytys ja joko hylätyt, tai hyväksytyt toimenpiteet. Lighthouse generoi sivustosta raportin. Erityisesti Lighthousen hylättyihin testeihin kannattaa kiinnittää huomiota sovellusta parannettaessa. Jokaisen tarkastuksen kohdalla on myös linkki dokumentaatioon, miksi kyseinen kohta on tärkeä ja miksi siihen tulisi kiinnittää huomiota. Lisäksi raportissa on mukana erityisesti kehittäjää kiinnostava tieto siitä, miten ongelman voi korjata. Lighthousea voi ajaa mitä tahansa nettisivua vasten joko Chrome-selaimen kehittäjätyökaluista, komentoriviltä tai NodeJs-moduulina. Lighthousen kehittäjätyökalujen asetuksia on esitelty kuvassa 4.3.



Kuva 4.3: Kuvakaappaus Google Chrome selaimen kehittäjätyökaluista: Lighthouse työkalun oletus asetukset.

Puppeteer on Node-kirjasto [44], joka tarjoaa korkean tason rajapinnan näkymättömässä tilassa ajettavaan Chrome-, tai Chromium-selaimeen kehittäjätyökalu rajapinnan kautta. Puppeteer-kirjastoa hyödyntämällä sovellus avattiin selaimessa useita kertoja kahdella eri tavalla. Ensimmäisessä ajossa selaimen annettiin tallettaa välimuistiin tietoa. Toisessa ajossa selain käynnistettiin aina uudestaan välimuisti tyhjennettynä. Näin saatiin tallennettua selaimelta kuluva aika sovelluksen kääntämiseen, jonka jälkeen selain on valmis vastaanottamaan käyttäjän syötteitä.

Ensimmäisenä mittarina käytetään Googlen Lighthouse-työkalua progressiivisuuden ja suorituskyvyn mittaamiseen. Lighthousen pisteytys on asteikolla 0-100. Toisena mittarina käytetään Puppeteer-kirjastolla tallennettuja sovelluksen kääntämiseen kuluvia aikoja. Mittaukset otetaan ilman Service Workeria, sekä Service Workerin välimuistin kanssa. Mittaustuloksia verrataan keskenään, josta nähdään kuinka paljon nopeammin Service Workerin välimuistista ladattu sivusto latautui. Lighthouse suosittelee lisäksi, että sovellusta testataan manuaalisesti useilla eri selaimilla ja alustoilla, jotta käyttökokemuksesta saadaan haluttu [26]. Manuaalitestauksessa sovellusta käytettiin Windows-työasemalla, macOS-kannettavalla, sekä Android-puhelimella.

4.4 Tutkimuksen eteneminen

Kirjallisuuskatsaus tuotti syksyllä 2018 ja keväällä 2019 rajallisen määrän akateemisia hakutuloksia. Hakusanoina käytettiin "progressive web app"ja "progressive web app service worker". Tietoa haettiin Google Scholarista kumpanakin ajankohtana. Haku tuotti 4-5 kappaletta järkeviä aiheeseen liittyviä tuloksia. IEEE Xplore tuotti saman määrän hakutuloksia ja ainakin yksi hakutulos oli molemmissa hakukoneissa sama. Hakusana "Service Worker"taas palautti suuremman määrän hakutuloksia, mutta aiheet eivät liittyneet tietotekniikkaan. Hakutuloksissa oli artikkeleita pääosin muilta aloilta.

Tutkimuksen suunnitteluvaiheessa sovelluksen arvioimisen laatumittareiksi valittiin nopeus ja progresiivisuus. Progressiivisuuden mittaamistyökaluksi valittiin Googlen Lighthouse-työkalu. Lighthouse antaa sovellukselle myös suorituskykypisteet, mutta ne eivät ole helposti verrannollisia keskenään, koska tulokset vaihtelevat hieman palvelimen ja testiä suorittavan koneen tehoista riippuen. PWA-sovelluksen nopeutta mitataan Puppeteer-kirjaston tallentamilla latausajoilla. Puppeteer-kirjasto valittiin nopeuden mittaamiseen, sillä rajapinnat vaikuttivat selkeiltä ja helppokäyttöisiltä. Puppeteer-kirjastoa hyödyntämällä latausaikoja voidaan tallentaa useita ja niistä voidaan laskea keskiarvo. Keskiarvoistettu data on luotettavampaa kuin Lighthousen yksitäinen ajo.

Toteutusvaiheessa sovelluksesta rakennetaan PWA-sovellus ja siihen liittyvät työkalut liitetään osaksi nykyisiä työkaluja. Valmista sovellusta mitataan välimuistitallennuksen ollessa käytössä ja ilman. Service Workerin vaikutuksia tarkastellaan Puppeteer-nopeusmittauksen tuloksista. Tuloksia havainnollistetaan kaavioilla ja taulukoilla. Progressiivisuutta ja PWA-sovelluksen kriteereitä tarkastellaan

Lighthouse-raporteista.

Tuloksista nähdään kuinka paljon nopeampi PWA-sovellus on. Työssä tutkitaan mitä Service Workerin käyttämisessä tulee ottaa huomioon ja millaisia heikkouksia tai haittoja Service Workerin käytöstä voi koitua.

5 Tekninen toteutus

Tässä luvussa esitellään yleisesti projektia ja sitä koskevaa arkkitehtuuria. Projektissa käytettyjä kirjastoja ja työkaluja esitellään lyhyesti. Luvussa käydään läpi, miten PWA-sovellus otettiin projektissa käyttöön.

5.1 Telia IoT-sovelluksen kehittäminen

Projektin teknologiaksi valikoitui Cumulocityn kehittäjäpaketin mukana toimitettava AngularJS:n versio 1.3. AngularJS on Googlen ylläpitämä avoimen lähdekoodin JavaScript-ohjelmistokehys, joka avustaa yksisivuisten sovellusten kehittämisessä ja käytössä [45]. AngularJS:n tavoite on lisätä selainpohjaisiin sovelluksiin tuki MVC-arkkitehtuurille, jonka avulla sivustojen kehitys helpottuu.

MVC-arkkitehtuurilla tarkoitetaan suunnittelumallia, jossa näkymä, malli ja ohjain on jaettu omiksi osioikseen [46]. Model, eli malli on ikäänkuin tietomalli, joka ratkoo pellin alla olevia ongelmia. Mallin ei tarvitse olla tietoinen ulkomaailmasta. View, eli näkymä, voi olla käyttöliittymä, rajapinta tai komentoriviohjelma. Controller, eli ohjain, kontrolloi puolestaan näkymää. Ohjain käsittelee syötteet ja näkymä tulosteet, kun malli huolehtii datasta.

AngularJS toteuttaa niin sanotun Single-Page-Application sovelluksen, eli SPA-sovelluksen. SPA-sovellus koostuu useista komponenteista, joita on mahdollista vaihtaa tai päivittää itsenäisesti ilman että koko sivu täytyy ladata uudestaan [47]. Tämä säästää latausaikaa, kun kaikkia kirjastoja ei tarvitse ladata uudestaan. SPA-sovelluksen tarkoitus on tehdä sovelluksesta modulaarinen, jolloin sovelluksen ylläpito ja kehitys helpottuu. Projektin loppupuolella AngularJS:n versio päivitettiin versioon 1.5.9.

5.2 Työkalut

Nykyiset JavaScript-pohjaiset käyttöliittymät ovat erittäin monimutkaisia. Koska kyseessä on pelkästään selaimella toimiva käyttöliittymä, on projektia laajennettu vuosien aikana useilla kirjastoilla. Sen ympärille on tuotu mukaan myös kehitystä helpottavia työkaluja.

Grunt on automaatiotyökalu [48]. Gruntin tarkoitus on tehdä toistuvia tehtäviä automaattisesti kehittäjän puolesta. Grunt toimii sovelluksen tärkeimpänä työkaluna, mutta pelkkä Grunt ei yksistään ole riittävä työkalu projektin tarpeille. Grunt huolehtii esimerkiksi sovelluksen kääntämisestä, paketoinnista, kirjoitusvirheiden tarkastamisesta, kuvien pakkaamisesta sekä web-palvelimen käynnistämisestä. Grunttiin on ladattu lukuisia laajentavia moduuleita.

Eslint on avoimen lähdekoodin projekti, joka on aloitettu vuonna 2013. Eslint on yhä edelleen suosituin linting-työkalu [49] Eslintin tarkoitus on toimia koodin tarkkailijana. Eslinttiin voi liittää lukuisia eri liitännäisiä, jotta koodista saa juuri ke-

hittäjälle mieluisan näköistä. Eslint pitää huolen esimerkiksi siitä, että koodilohkot on sisennetty kahdella välilyönnillä ja lohko päättyy aina puolipisteeseen. Mikäli kehittäjä ei noudata Eslinttiin kirjoitettuja sääntöjä, ei sovellusta käännetä ja kehittäjälle annetaan virheilmoitus virheellisestä syntaksista. Eslintin avulla koodia pyritään yhdenmukaistamaan, riippumatta siitä kuinka monta kehittäjää sovellusta on ollut tekemässä.

Yarn on riippuvuuksien hallinta työkalu [50]. Yarn toimii pohjana projektille. Yarnin avulla kaikki kehityksen aikaiseen toimintaan liittyvät skriptit ladataan kehittäjän koneelle. Myös sovelluksessa tarvittavat riippuvuudet ladataan Yarnilla. Yarn on Facebookin kehittäjien luoma työkalu. Sen tarkoitus on yhdenmukaistaa riippuvuuksien hallintaa, jolloin jokaisella kehittäjällä olisi käytössään juuri sama versio projektille määritellystä paketista. Yarn valittiin projektin riippuvuuksien hallintatyökaluksi lähinnä siksi, että projektin kehitystyön aikana siirryttiin pois Bowernimisestä Front-End riippuvuuksien hallintatyökalusta. Yarn osaa automaattisesti asentaa Bower-riippuvuuksia ja Bowerin sivuilla on suositeltu Yarnia sen korvaajaksi.

Less (Leaner Style Sheets) on laajennustyökalu perinteiselle CSS-kielelle [51]. Less tuo mukanaan muutamia hyödyllisiä laajennuksia, esimerkiksi muuttujat, operaatiot ja sisennykset. Muuttujilla voidaan esimerkiksi määrittää sovelluksen taustaväri, jonka jälkeen vaihtamalla muuttujan väriä saadaan sovelluksen taustaväri vaihtumaan jokaisella sivulla. Tyylien sisennyksellä taas on mahdollista kohdistaa tyylit vain tietyn muuttujan alle. Esimerkiksi hiiren hover efekti voidaan laittaa suoraan alkuperäisen tyylin sisälle, ilman että sitä varten tarvitsee luoda uutta riviä.

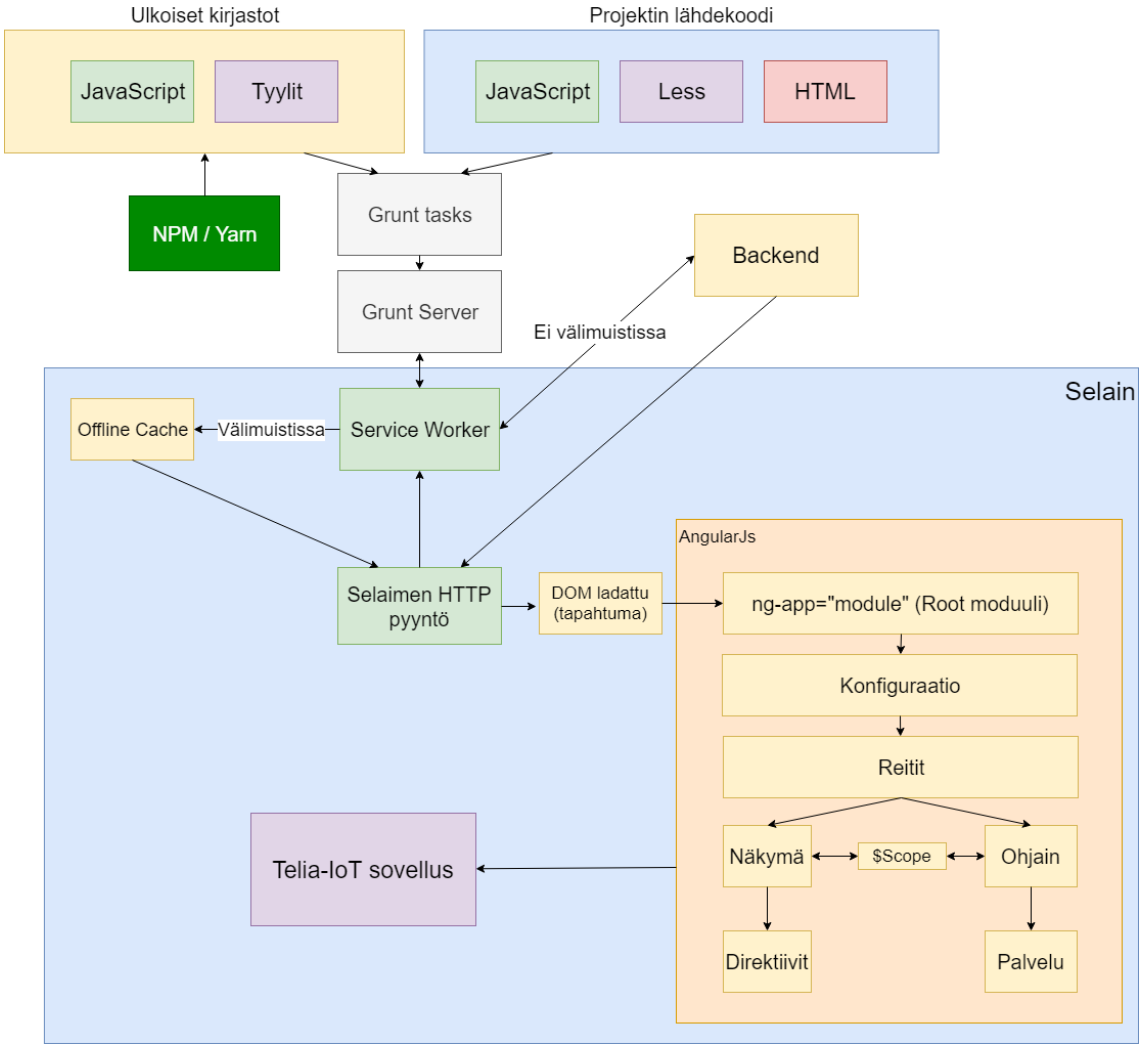
Babel on uuden sukupolven JavaScript-kääntäjä [52]. Babelia käyttämällä sovelluskehittäjä voi kirjoittaa tuoreinta mahdollista JavaScript syntaksia, ilman että selainvalmistajat ovat implementoineet uusia toimintoja selaimeen. Babelin avulla uusi JavaScript syntaksi käännetään vanhojen selaimien ymmärtämään JavaScript muotoon. Babelia on myös mahdollista laajentaa liitännäisillä. Sitä mukaa kun selaimien JavaScript tuki laajenee, voidaan Babelista poistaa liitännäisiä kehityksen jatkuessa.

Bootstrap on tehokas käyttöliittymäkirjasto, jota käyttämällä on helppoa luoda responsiivisia, mobiili edellä kulkevia projekteja [53]. Kirjasto tuo mukanaan esimerkiksi responsiivisen Grid-systeemin ja lukuisia käyttövalmiita komponentteja, kuten kuvakarusellin ja tyylitellyt taulukot. Kirjaston käyttäminen vähentää CSS:n parissa tapahtuvaa työskentelyä, sillä käyttämällä valmiita muuttujia sovelluksesta saadaan responsiivinen, joka näyttää oikealta kaiken kokoisilla näytöillä.

Webpack. Joulukuussa 2018 projektiin asennettiin Gruntin tilalle Webpack, joka on modulaarinen paketointityökalu nykyaikaisille web-sovelluksille [54]. Webpackin avulla kirjastoja voidaan ladata modulaarisesti projektin käyttöön. Ennen kirjastoja ylläpiti kehittäjä lisäämällä <script> tagin aina HTML-sivun alkuun. Nyt Webpack huolehtii moduulien lataamisesta. Ainoa työvaihe on lisätä moduulien lataaminen automaatiotyökaluun, Webpackkiin, joka valitsee oikeat moduulit. Sovelluksen kokoa saadaan pienennettyä valitsemalla moduuleista ladattavat palaset. Tämä vaikuttaa

suoraan sovelluksen nopeuteen.

Sovellusta käännettäessä JavaScript-koodin toimenpiteiden jälkeen CSS-tyylitiedostot käännetään LESS-formaatista selaimen ymmärtämään CSS-muotoon. Kaikki tyylitiedostot paketoidaan myös yhdeksi kokonaisuudeksi ja sitten ne pienennetään automaattisella työkalulla. Lisäksi suoritetaan lukuisia muita toimenpiteitä, projektin kaikki kuvat pakataan, kolmannen osapuolen skriptit käydään läpi, paketoidaan ja liitetään projektiin. Sovellus koostuu ulkoisista kirjastoista, sekä lähdekoodista. Selain tekee kutsuja joko taustapalveluihin, tai Service Workerin kautta lokaaleihin tiedostoihin. Sen jälkeen selain evaluoi skriptit ja kääntää ne näkymäksi. Sovelluksen arkkitehtuuria on havainnollistettu kuvassa 5.1.



Kuva 5.1: Telia IoT-sovelluksen arkkitehtuuri.

5.3 PWA-sovelluksen käyttöönotto projektissa

PWA-sovelluksen kehittäminen aloitettiin lisäämällä projektiin sopiva sovelluskuvake ja nimi Web App-manifestiin. PWA-sovelluksen vaatimusten mukaisesti projektiin piti lisätä Service Worker, mikä oli isompi ja haastavampi kokonaisuus. Sovellusta päätettiin laajentaa Googlen tarjoamalla Workbox CLI-työkalulla. Workbox on NodeJS-pohjainen komentoriviohjelma, jolla on helppoa luoda Service Worker nykyiseen olemassa olevaan projektiin. Lisäksi ohjelma tarjoaa käteviä komentoja, joilla toimintaa voidaan mukauttaa omaan projektiin sopivaksi. Workbox asennettiin projektiin käyttämällä Yarnia.

Workboxin käyttöönotossa huomattiin, että Workbox ei ole tuettuna Grunt-työkalussa, sillä Grunt on liian vanha. Workboxia on kuitenkin mahdollista suorittaa erikseen myös NodeJs-skripteillä, joten Grunt-työkalua laajennettiin Grunt-run-nimisellä paketilla, joka mahdollistaa komentojen suorittamisen Gruntissa.

Gruntiin luotiin run-tehtävä listauksessa 5.1, jonka tarkoitus oli ajaa Workbox-skriptit. Tärkeää run-komennossa on liittää argumenteiksi “inject:manifest“ [55], jolloin ohjelma tietää liittää lopulliseen Service Worker -tiedostoon kaikki tiedostot, jotka pitää lisätä välimuistiin. Näin jokaista tiedostoa tai polkua ei tarvitse kirjoittaa itse. Riittää, että konfiguraatiotiedostossa on määritelty välimuistiin tallennettavien tiedostojen päätteet. Injektointikomento on Workboxin mukana tuleva natiivi komento.

```
module.exports = {
  options: {},
  generatesw: {
    cmd: './node_modules/workbox-cli/build/bin.js',
    args: ['inject:manifest']
  }
};
```

Listaus 5.1: Grunt run-komento.

Kun Grunt on ajanut sovelluksen rakentamiseen liittyvät tehtävät, ajaa se lopussa run-tehtävän, joka taas ajaa Workbox-skriptin. Skripti käyttää projektin juureen generoitua Service Worker konfiguraatiotiedostoa, joka saatiin alun perin käyttämällä Workboxissa toimitettua asennusvelhoa. Workboxin asennusvelho on automaattinen työkalu konfiguraation luomiseen. Konfiguraatiossa määritellään esimerkiksi, mistä kansiosta tiedostoja otetaan mukaan Service Workeriin ja millä tiedostopäätteellä olevat tiedostot lasketaan mukaan. Esimerkkinä IoT-projektin Service Worker konfiguraatio listauksessa 5.2.

```
module.exports = {
  globDirectory: 'dist/',
  globPatterns: [
    '**/*.{css,otf,ttf,eot,woff,woff2,svg,png,html,js,json}'
  ],
```

```
  swSrc: './src/js/service-worker.js',
  swDest: './dist/sw.js',
  globIgnores: [
    '../workbox-cli-config.js',
    'vendor/bowerbundle.js'
  ]
};
```

Listaus 5.2: Service Worker konfiguraatio IoT-projektissa.

Projektissa Service Worker pohja, joka on määritetty konfiguraatiotiedostossa. Tähän varsinaiseen Service Worker -tiedostoon Workbox osaa konfiguraation pohjalta liittää tarvittavat tiedostot. Listaus 5.3 on Service Worker-pohja, jota Workbox laajentaa automaattisesti injektiokomennolla.

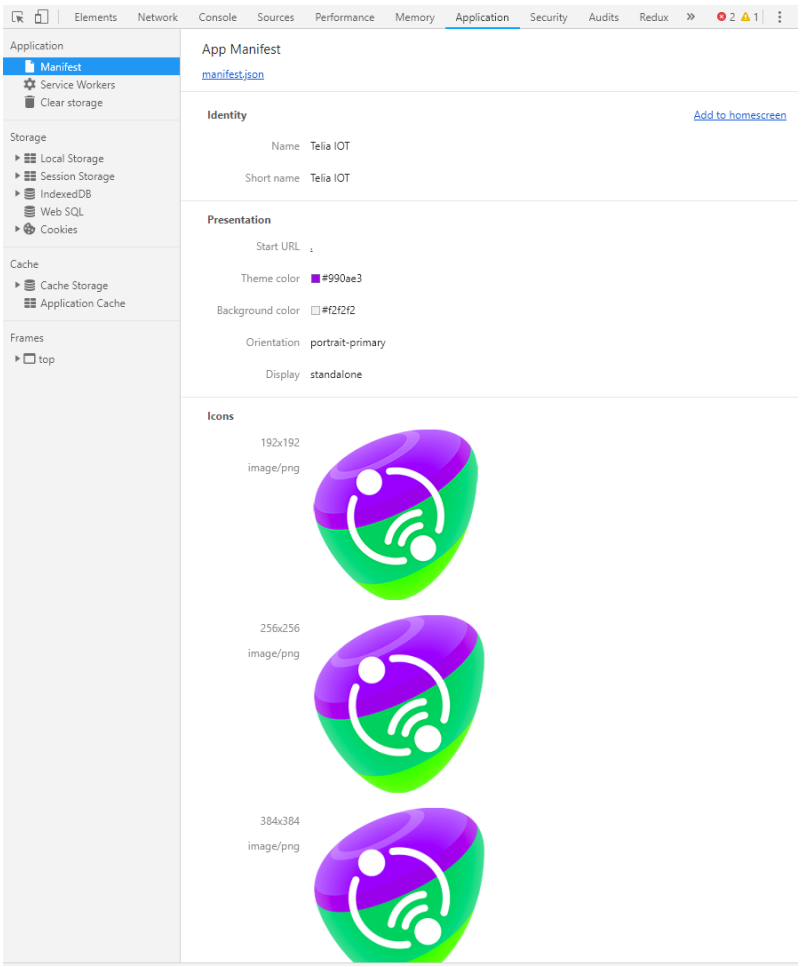
```
workboxSW.precache([]);

const cacheOneWeekStrategy = workboxSW.strategies.cacheFirst({
  cacheName: 'cdn-cache',
  cacheExpiration: {
    maxEntries: 20,
    maxAgeSeconds: 7 * 24 * 60 * 60, // one week
  },
  cacheableResponse: {
    statuses: [0, 200],
  },
});

workboxSW.router.registerRoute(
  /https:\/\/fonts.googleapis.com\/(.*)/,
  cacheOneWeekStrategy
);

workboxSW.router.registerRoute(
  '/',
  workboxSW.strategies.networkFirst()
);
```

Listaus 5.3: Service Worker-tiedoston runko mitä Workbox laajentaa.

Service Workerissa on määritelty workboxSW precache-funktio, jo nka parametrina on tyhjä taulukko. Taulukkoon Workbox laajentaa konfiguraatiossa mainitut tiedostot, jotta ne voidaan ottaa Service Workerin välimuistissa talteen. Service Workeriin on myös määritelty yhden viikon välimuististrategia, ja esimerkiksi Googlen tekstityypit on lisätty tähän yhden viikon strategiaan. Tekstityypit eivät päivity kovin usein ja siksi on hyödyllistä laittaa niille hieman pidempi välimuistitallennus. Itse varsinaiselle sovellukselle taas on laitettu strategiaksi verkko edellä. Koska sovellus

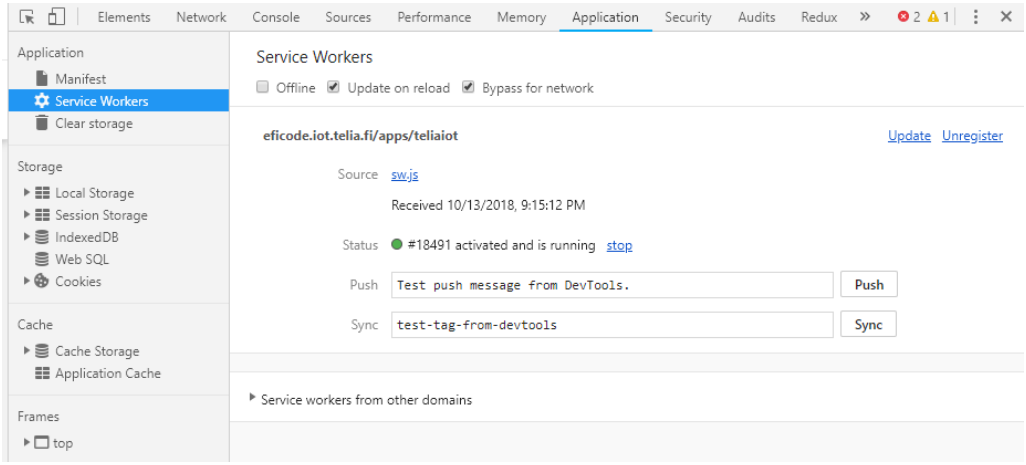
tarvitsee käytännössä aina verkkoyhteyden datan hakemiseksi, ei ole järkevää asettaa strategiaksi välimuistia, vaan käyttäjälle halutaan aina näyttää ensiksi verkosta saatavilla oleva uusin tieto. Vasta verkkoyhteyden ollessa poissa käytöstä, voidaan välimuistista näyttää tietoja.

Manifestin toimintaa sovelluksessa voitiin tarkastella Google Chrome -selaimen kehittäjäkonsolista sovellusvalikon alta löytyvältä Manifest-välilehdeltä, esitelty kuvassa 5.2. Välilehdeltä selviää heti toimiiko manifesti oletetulla tavalla. Mikäli määrittelyssä on virheitä ne ilmoitetaan tällä sivulla. Sovelluksen identiteetti tiedoissa tulee lukea manifestin vaatimuksissa määritellyt kentät: sovelluksen nimi, sekä lyhytnimi. Presentaatiokohdassa tulee olla määriteltynä sovelluksen käynnistysosoite, teeman väri, sovelluksen orientaatio mobiililaitteella sekä näyttömuoto. Ikoni-kentissä tulee olla sovelluksen ikoni, joka näytetään käyttäjän mobiililaitteella asennuksen jälkeen. Ikoneita tulee olla useammassa eri resoluutiossa.



Kuva 5.2: Manifest.json-tiedoston sisältö ja toimivuus tarkasteltuna Google-Chrome kehittäjäkonsolissa.

Google Chromen kehittäjäkonsolin sovellusvälilehdeltä löytyy myös Service Worker -ikkuna, josta voidaan tarkistaa Service Workerin asentuminen, esitelty kuvassa 5.3. Välilehdeltä selviää Service Workerin lähdeskripti sekä tila. Mikäli Service Worker on tilassa “activated and running“ on se asentunut oikein ja käytössä. Service Workerin yhteyteen on myös liitetty valinnat “Offline“ “Update on reload“ ja “Bypass for network“ -valintaruudut. "Update on reload"ja "Bypass for network"onkin järkevää asettaa päälle, kun sovellusta ollaan kehittämässä. Muuten sovellus saattaa näyttää kehittäjälle vanhaa välimuistissa olevaa tietoa. Valintaruutujen toiminta on loogista, Service Worker päivitetään aina sivuston latautuessa uudestaan “Update on reload“ -valinnan ollessa päällä ja välimuisti saadaan ohitettua “Bypass for network“ -valinnalla. "Bypass for network-valinta tulee kuitenkin ottaa pois päältä, kun välimuistia ja sovelluksen nopeutta halutaan taas testata.



Kuva 5.3: Service Workerin asetukset Google-Chromen kehittäjäkonsolissa.

Joulukuussa 2018 projektin kehitystyökaluja haluttiin päivittää. Projektin kasaava Grunt-työkalu on kehitetty vuonna 2012 ja se haluttiin korvata yksinkertaisemmalla konfiguraatiolla ja samalla siirtyä nykyaikaiseen modulaariseen kehitykseen. Nykyinen malli ei sallinut uusien JavaScript-pakettien lataamista ja niiden tuomista projektin käyttöön helposti. Uudeksi build-työkaluksi valittiin Webpack, joka on tarkoitettu modulaarista ja nykyaikaista kehittämistä varten. Paketinhallintatyökaluksi päätettiin myös vaihtaa jälleen NPM:n, eli Node Package Manager, joka on ottanut Yarnin kehityksessä kiinni.

Työkalujen päivitys aloitettiin siirtymällä Yarn-työkalusta NPM:n käyttöön. Suurin osa paketeista asentui NPM-työkalulla suoraan. Yarnilla erityisesti asennettavia pakettaja olivat erittäin vanhasta Bower-työkalusta asennetut kirjastot, jotka joko korvattiin uusilla NPM-rekisteristä löytyvillä paketeilla, tai asennettiin suoraan Githubin URL-osoitteella.

Seuraava vaihe oli vaihtaa vanha 500 rivinen konfiguraatio Webpackille. Webpack on modulaarinen paketointityökalu. Moduulien ansiosta osa skripteistä pystyttiin lataamaan vain tarvittaessa. Ongelmaksi muodostui kuitenkin se, että osa skrip-

teistä oli liian vanhoja ja niitä ei ollut mahdollista ladata moduuleina. Kun sovellus oli saatu kääntymään Webpackilla vanhoja skriptejä lukuun ottamatta, otettiin sovelluksesta uudet mittaukset. Lopullisesta Webpack konfiguraatiosta tuli noin 120 riviä, eli huomattavasti yksinkertaisempi. Lisäksi Webpack toi mukanaan lukuisia uusia hyödyllisiä toimintoja. Nyt uusia kirjastoja ja moduuleita voitaisiin ottaa käyttöön suoraan NPM-rekisteristä, sekä Webpackin myötä sovelluksen kokoa saataisiin hieman pienennettyä. Sovelluksen optimointiin ei käytetty kuitenkaan paljon aikaa. Tärkeimpänä uudistuksena saatiin päivitettyä Googlen Workbox kirjasto, jolla PWA-sovelluksille vaadittava Service Worker skripti saataisiin luotua automaattisesti. Listauksessa 5.4 Webpackille luotu Service Worker konfiguraatio.

```
new GenerateSW({
  skipWaiting: true,
  swDest: 'sw.js',
  include: [/\.html$/, /\.js$/, /\.png$/, /\.jpg$/],
})
```

Listaus 5.4: Webpack konfiguraatio Service Workerin luomiseksi.

Webpack luo Workbox lisäosalla automaattisesti sopivan Service Worker scriptin, kun Webpack käynnistetään. Parametreiksi on asetettu skipWaiting, joka pakottaa odottavan Service Workerin aktivoitumaan, Service Workerin nimeksi on asetettu sw.js ja etukäteisvälimuistiin talletettaviksi tiedostoiksi on määritelty kaikki .html, .js, .png ja .jpg päätteiset tiedostot. Näillä tiedoilla Service Worker generoituu automaattisesti ja sen välimuistissa on valmiiksi sovelluksen tiedostot. Aikaisempi usean rivin konfiguraatio saatiin nyt mahtumaan vain neljälle riville.

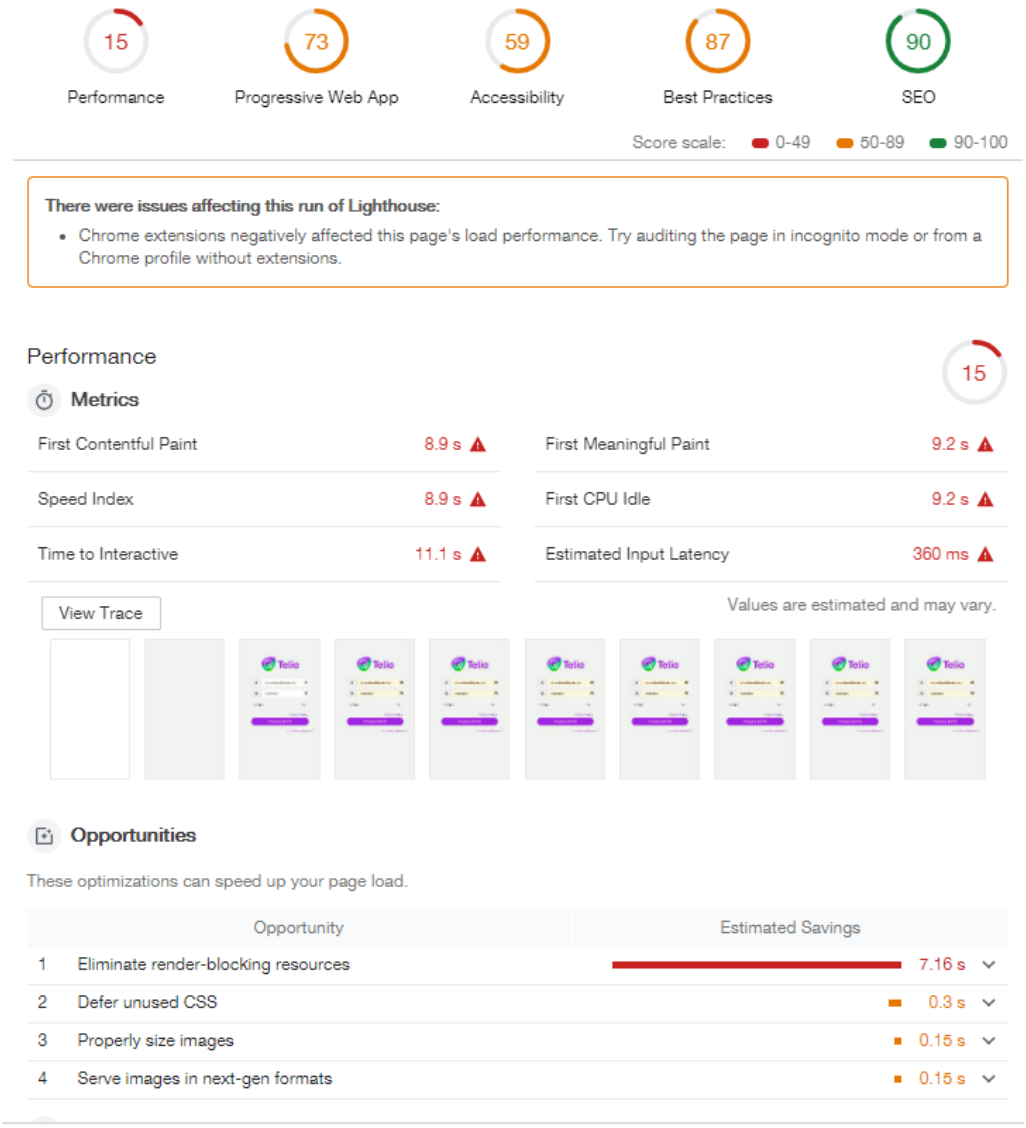
6 Arviointi

Tässä luvussa arvioidaan PWA-sovelluksen nopeutta verrattuna normaaliin verkkosivustoon. Sovellusta mitattiin suoraan tuotantopalvelimelta. Vaihtoehtona olisi ollut mitata sivua myös kehittäjän koneelta paikallisesta kehitysympäristöstä, mutta tällöin ei olisi saatu tarpeeksi kattavia tuloksia loppukäyttäjän näkökulmasta. Ensiksi mitattiin sovelluksen versiota 1.7.0, joka oli kasattu vanhalla Grunt-työkalulla, sekä vanhalla Service Worker-skriptillä. Seuraavaksi mitattiin sovelluksen versiota 1.7.1, jossa oli toteutettuna uusi Service Worker ja Webpack.

6.1 Lighthousen progressiivisuuspisteet ja suorituskyky

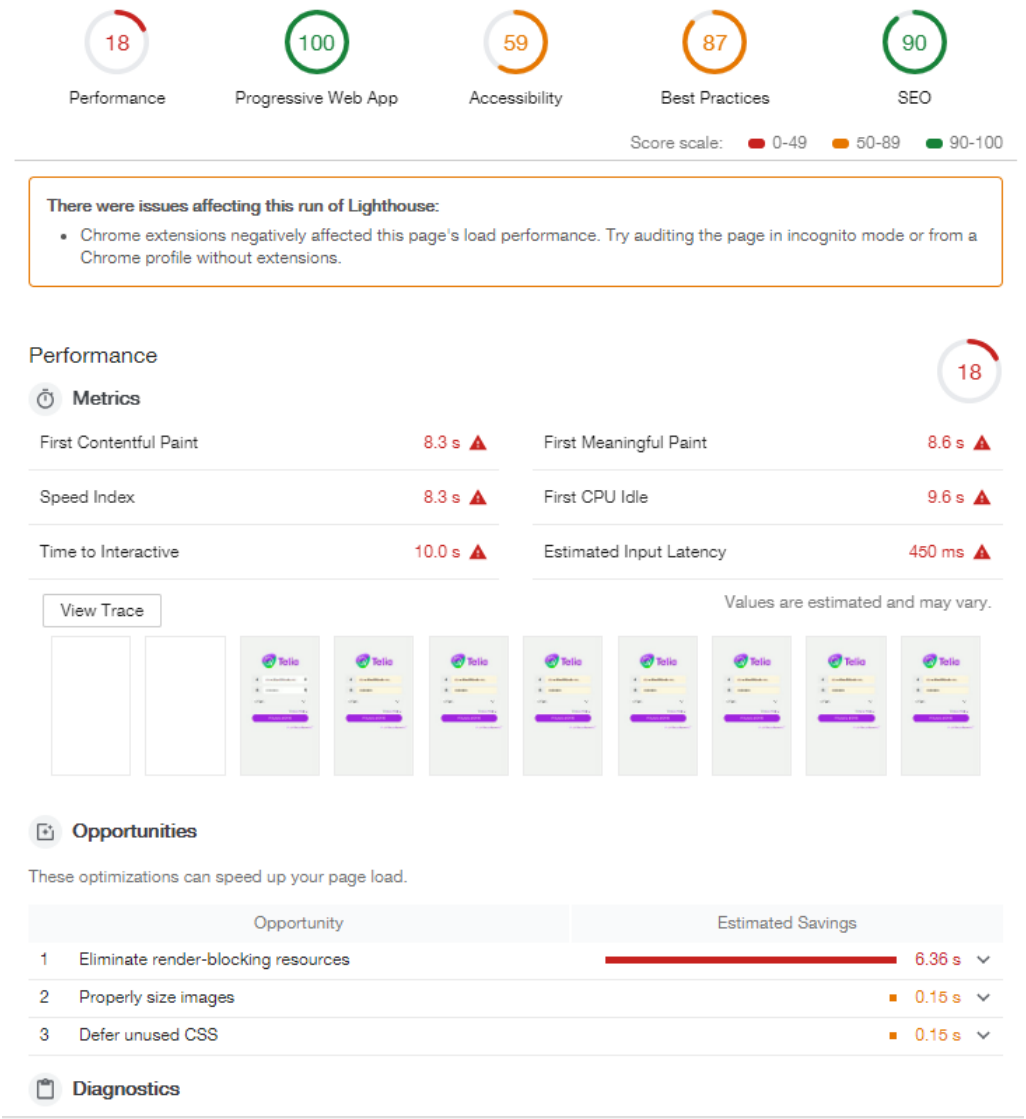
Sovelluksen koko on kasvanut vuosien aikana erittäin laajaksi lukuisten kirjastojen ja kirjoitetun koodin myötä. Isomman koodimäärän lataaminen ja kääntäminen selaimessa kestää kauemmin. Vuoden 2015 lopulla aloitettu projekti ei ole modulaarinen ja sen skriptien lataaminen on selaimessa hidasta. Lisäksi kaikki skriptit ladataan, vaikka niitä ei edes heti tarvittaisi. Tästä syystä suorituskykypisteet jäivät hyvin pieniksi ensimmäisessä mittauksessa. Suorituskykypisteitä verrattiin Lighthouse-työkalun mittapisteestä: aika joka selaimelta kesti kääntää sovellus niin, että se on valmis käytettäväksi ja reagoi käyttäjän syötteisiin.

Toinen pisteisiin suoraan vaikuttava asia on ensimmäinen järkevä piirto-mittapiste, joka tarkoittaa ensimmäistä järkevää näkymää käyttäjälle. PWA-sovelluksille on erittäin tärkeää latautua nopeasti. Lighthousen tuloksista, kuva 6.1 nähdään, että sovellus täyttää kaikki tarvittavat PWA-sovelluksen kriteerit, paitsi suorituskyvyn osalta, jonka takia PWA-pistemäärä jäi 73 pisteeseen.



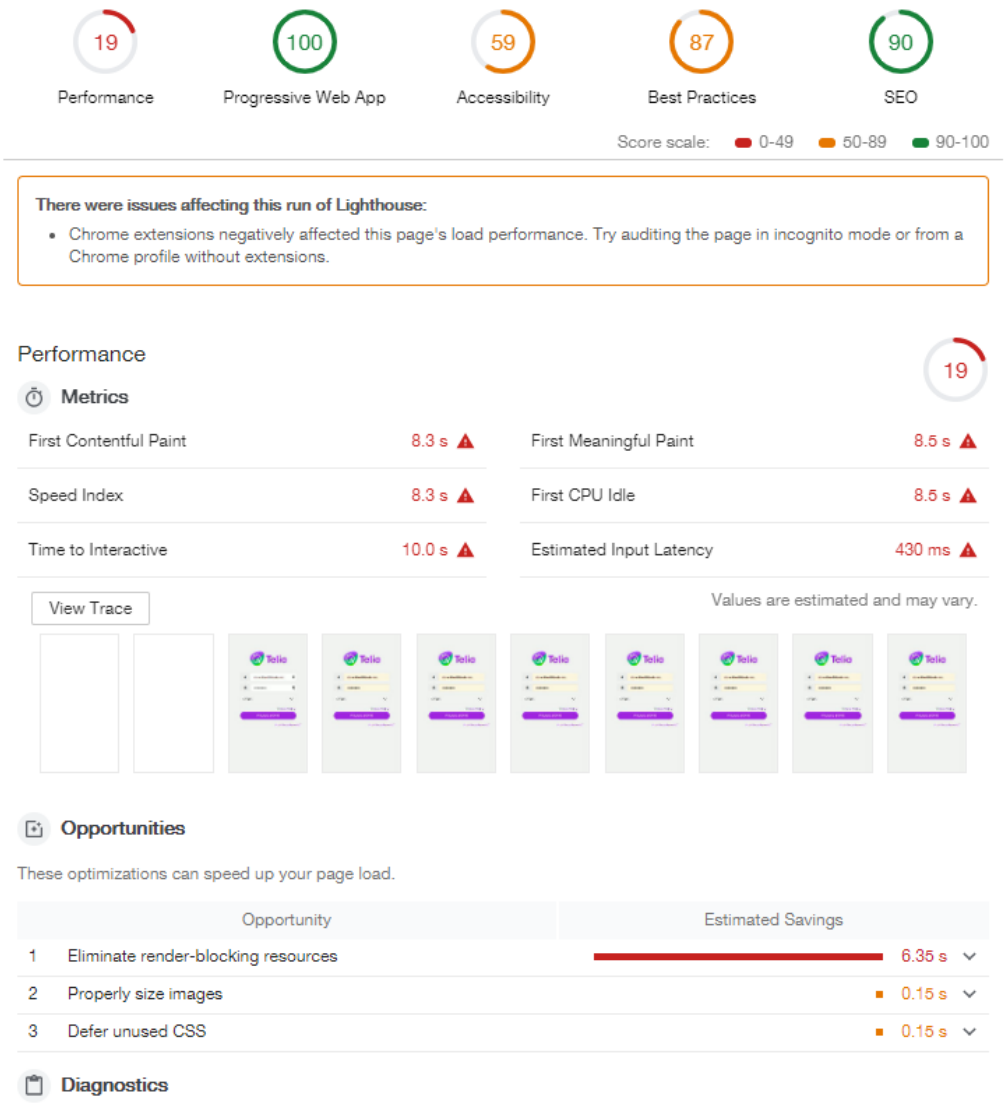
Kuva 6.1: Lighthouse raportti Grunt-työkalulla paketoidusta sovelluksesta vanhalla Service Worker skriptillä.

Seuraava mittaus tehtiin Webpackia hyödyntävällä sovelluksen versiolla 1.7.1, jonka Service Worker oli generoitu Webpack-lisäosalla. Pelkästään siirtämällä muutama skripti ladattavaksi moduuleina, saatiin sovelluksen lataamista nopeammaksi. Mittauksen tuloksista kuvassa 6.2 nähdään, että sovelluksen kääntöaika käytettäväksi laski yli sekunnin. Myös ensimmäinen järkevä piirto tippui yli puoli sekuntia. Websovelluksista puhuttaessa on kyse huomattavasta ajasta. Sovelluksen kokonaiskäynnistymisaika on kuitenkin edelleen hidas verrattuna hyvin optimoituihin nykyaikaisiin modulaarisiin sivustoihin. Sekunnin nopeusmuutos antoi kuitenkin täydet 100 pistettä PWA-kategoriasta.



Kuva 6.2: Lighthouse raportti Webpack-työkalulla paketoidusta sovelluksesta uudella Workboxin generoimalla Service Worker skriptillä.

Sovelluksen nopeutta mitattiin vielä kolmannen kerran, jotta saataisiin tulokset Service Workerin muodostamasta välimuistin käytöstä. Kolmannella kerralla mitattaessa Lighthousesta laitettiin "Clear storage"valinta pois päältä, jotta sovelluksen välimuistia ei tyhjennettäisi. Sovellus sai toisesta auditoinnista vielä yhden pisteen lisää suorituskykyyn. Ensimmäinen järkevä piirto tippui vain 0.1 sekuntia. Tulokset ovat kuvassa 6.3.



Kuva 6.3: Lighthouse raportti Service Workerin cachen kanssa.

6.2 Puppeteer mittausten-suorituskyky

Jotta Service Workerin vaikutuksista saataisiin vielä tarkempia tuloksia tehtiin seuraavaksi yksinkertainen toistuva sivulataus sovellukselle. Sovellus avattiin selaimessa

ja siitä tallennettiin dokumenttioliomallin kääntöaika. Tällöin selain on parsinut päädokumentin, mutta sovellus ei ole vielä käytettävissä eikä reagoi syötteisiin. Sovellusta vasten ajettiin yksinkertainen rasitustesti kahdella eri tavalla. Ensimmäiseksi selain käynnistettiin joka kerta uudestaan, jotta selaimen välimuistissa ei olisi aikaisempaa tietoa. Seuraavaksi selainta ei käynnistetty uudestaan ja sivulataus tehtiin, kun Service Workerin välimuistiin oli talletettu tietoa.

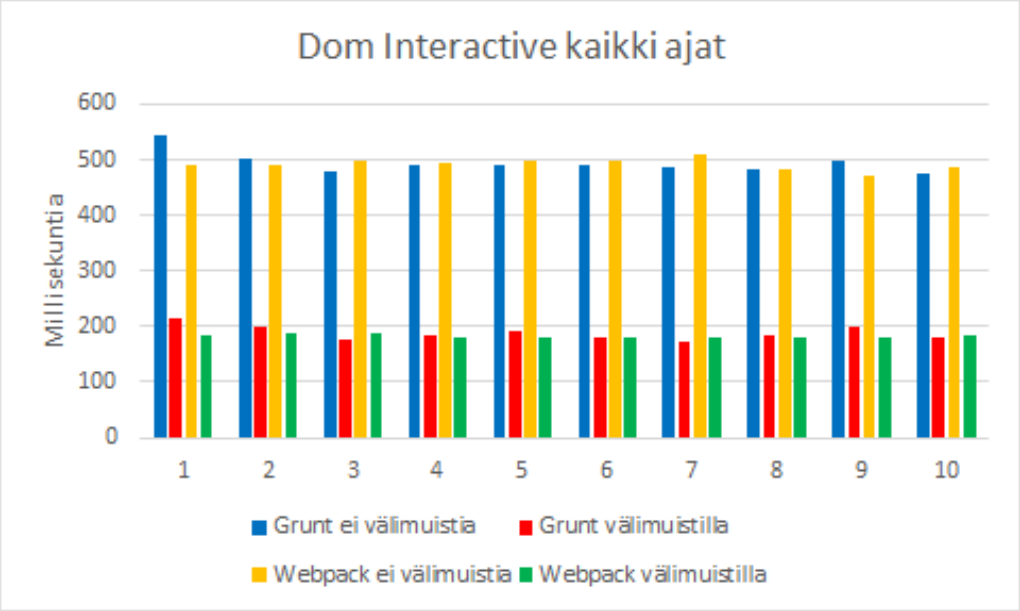
Testissä määriteltiin yksinkertaisesti kymmenen ajoa kummallekin tavalle. Selaimen objekteista löytyy suorituskyvyn ajoitustieto, josta saadaan tietoa sovelluksen vasteajoista. Testeihin määriteltiin kerättäväksi edellä mainittu dokumenttioliomallin kääntämisen aika. Jälkimmäisessä testitapauksessa, jossa selainta ei käynnistetty uudestaan, testit ajettiin 11 kertaa. Ensimmäistä ajokertaa ei huomioitu, sillä silloin välimuistissa ei vielä ollut mitään tietoa sivusta. Puppeteer aloittaa ajon aina oletuksena tyhjältä pöydältä. Tuloksia on esitelty taulukossa 6.1 ja kuvassa 6.4.

Grunt ei Välimuistia	**Grunt Välimuistilla**	**Webpack ei Välimuistia**	**Webpack Välimuistilla**
544	215	492	183
501	200	490	189
479	176	500	189
491	183	494	181
491	192	497	179
492	179	498	182
487	171	510	180
482	184	482	179
500	199	470	182
475	181	486	184
Keskiarvo			
494,2	188	491,9	182,8

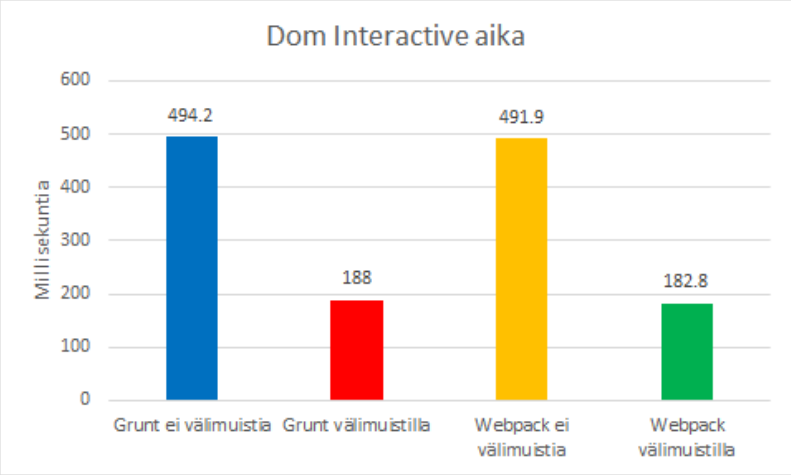
Taulukko 6.1: Telia IoT-palvelun latausajat testissä.

Service Workerin välimuistista ladattu sivu aukeaa huomattavasti nopeammin kuin ilman Service Workeria. Pyöristettynä Service Workerin välimuistista ladattava sovellus aukeaa 300 millisekuntia nopeammin. Keskiarvoistetut ajat on havainnollistettu kuvassa 6.5. Latausaika tuntui vaihtelevan hieman suuntaan tai toiseen. Laskemalla keskiarvo kummastakin ajosta nähdään, että Webpackilla tehty ratkaisu oli lopulta hieman nopeampi. Pieni ero Webpackin ja Gruntin välillä johtui siitä, että sovellusta ei oltu optimoitu tarpeeksi modulaariseksi, eikä ylimääräisiä kirjastoja jätetty lataamatta uudessa Webpack-versiossa. Tätä ei tehty siksi, että käytetyt kirjastot olivat joko liian vanhoja ladattavaksi modulaarisesti, tai niitä ei ehditty päivittää uudempiin.

Lighthouse, sekä Puppeteer-testin perusteella Service Workerin välimuistia hyödyntämällä sovelluksen latausaikaa saadaan siis pienennettyä huomattavasti ja sen käyttäminen on järkevää.



Kuva 6.4: Dokumenttioliomallin kääntämisen kaikki ajat.



Kuva 6.5: Dokumenttioliomallin kääntämisen keskiarvoistettu aika.

6.3 Pohdinta

Työssä päätettiin jättää vertailu hybridi- ja natiivisovellusten kesken pois, sillä sovelluksen muuttaminen kyseisille teknologioille sopivaksi olisi ollut liian haasteellista tai aikaa vievää.

PWA-sovelluksen käyttöönotossa esiintyi muutamia ongelmia projektin aikana. Projektin työkalut olivat vanhentuneet, joten ensimmäinen varsinainen ongelma oli yhteensopivuusongelma Service Workerin luomisessa. Toinen kehityksen aikana ilmennyt ongelma oli sovelluksen näyttäytyminen välimuistista, kun Service Worker oli saatu käyttöön. Google Chromen kehittäjäkonsolissa on Service Worker -välilehdellä valinnat “update on reload“ ja “Bypass for network“ jotka ruksattiin käyttöön. Näin kehittäjä näkee aina uusimmat muutokset selaimessaan. Muutoin koodiin tehdyillä muutoksilla ei välttämättä olisi vaikutusta, sillä Service Worker näyttäisi tiedostoja ja koodia välimuistista eikä suoraan paikalliselta kehityspalvelimelta.

"Bypass for network-valinnan käytöstä aiheutui kuitenkin, että kehityksen aikana ei voitu hyödyntää Service Workerin välimuistia kun välimuistin käyttö ohitettiin. Välimuistin käyttö kehityksen aikana vähentäisi sovelluksen latausaikaa, mikä olisi kehittäjälle hyödyllistä.

Kolmas haaste liittyi sovelluksen modulaarisuuden puuttumiseen. Nykyisessä sovelluksessa on paljon koodia, joka suoritetaan ohjelman käynnistyessä, vaikka koodia ei vielä tarvittaisi. Tästä syystä Lighthousen auditointityökalu antoi huonot suorituskykypisteet sovellukselle. Koodien pakkaamiseen tehtiin joitakin muutoksia työkalujen käyttömahdollisuuksien mukaan jolloin tilannetta saatiin hieman parannettua. Myös tietokantakyselyitä koetettiin vähentää. Kyselyiden vähentäminen on kuitenkin haastavaa palvelussa, joka perustuu reaaliaikaiseen dataan.

Testit osoittavat selvästi, että välimuistin käyttö nopeuttaa sovelluksen toimintaa. Puppeteer-testeissä eivät kuitenkaan käyneet ilmi kaikki mahdolliset käyttötapaukset. Esimerkiksi tilanne, jossa Service Worker on sammunut selaimessa. Selain siis saattaa sammuttaa Service Workereita, jos sivustolla ei ole vierailtu vähään aikaan ja Service Workereita on kertynyt useampia. Tällöin Service Workerin käynnistämisestä seuraa pieni vasteaika, joka vaikuttaa suoraan sivuston latausaikaan. Googlen tapaustutkimuksessa on kuitenkin laskettu, että Service Workerin käynnistymisaika on vähintään 100 millisekuntia mobiililaitteilla ja tietokoneilla 20-100 millisekuntia. Service Workerin käyttö ei siis ole ilmaista. Tästä huolimatta kaikki sivulataukset olivat nopeampia Googlen tapaustutkimuksessa, jos Service Worker oli käytössä ja vaikka se olisi sammunut. Googlen tapaustutkimuksessa vertailtiin myös selaimen omaa välimuistin käyttöä Service Workeriin. Jokaisessa tapauksessa Service Worker oli nopeampi.

Service Workerin ongelma on välimuistin hallinta. Kehittäjän tulee miettiä tarkkaan, mitä tietoa kannattaa ja voi laittaa välimuistiin. Kaikkia verkkokaupan kuvia ei esimerkiksi ole järkevää laittaa välimuistiin. Huolellisella välimuistisuunnittelulla, ja käyttämällä Service Workerin kanssa erilaisia strategioita, saadaan aikaan järkevä kokonaisuus. Koska tallennustila on rajallinen, esimerkiksi mobiililaitteita varten

nyrkkisääntönä voidaan varata noin 50 megatavua tilaa välimuistille [23].

Itse PWA-sovelluksen luominen olemassa olevaan sovellukseen on loppujen lopuksi yksinkertainen prosessi. Sovelluksen tarvitsee ainoastaan rekisteröidä Service Worker ilman että se edes tekisi mitään. Lisäksi tarvitaan yksinkertainen JSON-manifesti, jossa on määritelty sovelluksen nimi, sovelluskuvake ja muut tarvittavat tiedot. Tiedostoja pystyy jopa generoimaan verkossa, jos sellaista ei halua kirjoittaa itse. Lisäksi tarvitaan sovelluksen kuvakkeet eri resoluutioissa ja sovellus tulee tarjota HTTPS-protokollan ylitse.

PWA-sovelluksen toteuttaminen automatisoiduilla työkaluilla on nopeaa ja edullista. Service Worker nopeuttaa sovelluksen latausaikaa mikäli välimuististrategia on mietitty ja toteutettu oikein. Jos välimuististrategia on liian agressiivinen, ja kaikki mahdollinen tieto halutaan tallettaa välimuistiin, se saattaa hidastaa sovelluksen toimintaa.

Tämän tutkimuksen ja Googlen tapaustutkimuksen lopputuloksena:

- Sivut latautuivat keskimäärin huomattavasti nopeammin Service Workerin ollessa käytössä, jos sen välimuistissa oli jo jotain.
- Vierailut sivuille, jotka Service Worker oli ladannut välimuistiin, aukesivat melkein heti.
- Googlen tutkimuksessa huomattiin, että mikäli Service Worker on sammuneena selaimessa, kestää sillä pieni hetki käynnistyä. Tästä huolimatta sivusto, jolla Service Worker oli käytössä ja sammuneena latautui silti nopeammin kuin sivu, jolla ei ollut ollenkaan Service Workeria.
- Mobiililaitteilla Service Workerin käynnistyminen kestää kauemmin kuin tietokoneella.

Huomioon otettavaa on myös, että mikäli Service Workerin ja PWA-sovelluksen suorituskykyä halutaan mitata, tulee suunnitella jokaista sovellusta varten oma mittausstrategia. Minkälaisia arvoja ja mittapisteitä halutaan ottaa huomioon? Jos esimerkiksi mittapisteenä halutaan käyttää "ensimmäinen piirto"arvoa, tulee huomioida ensimmäinen järkevä piirto. Joissain tapauksissa selain saattaa ilmoittaa ensimmäisen piirron arvon tyhjälle sivulle. Sivu voi olla tyhjä, riippuen tyylien ja sisällön latustavasta. Tärkeää on tässä tapauksessa määritellä, mikä on ensimmäinen järkevä näkymä sivulla.

Service Workerin välimuistiin oli määritetty tallennettavaksi kaikki sovelluksen skriptit, kuvat ja näkymät etukäteen. Lisäksi olisi ollut mahdollista lähettää myös sovellusilmoituksia, mutta siihen ei ollut tarvetta. Sovellusta ei voi käyttää ilman verkkoyhteyttä, sillä suurin osa näkymistä perustuu reaaliaikaiseen tietoon. Ilman verkkoyhteyttä sovellus on mahdollista avata, mutta käyttäjä ohjataan kirjautumissivulle. Mahdollisuutena verkkoyhteydettömässä käytössä olisi ollut tallettaa datakutsuja Service Workeriin ja näyttää vanhoja mittapisteitä ilmanlaadusta. Tätä ei keretty projektissa toteuttamaan.

Yksi mielenkiintoinen PWA-sovelluksien käyttökohde on älytelevisiot. Älytelevisioiden sovelluskehitys on hyvin pirstoutunutta tällä hetkellä. Jokaisella laitevalmistajalla on oma käyttöliittymä ja ohjelmistokehys. Lisäksi televisioita voidaan laajentaa erilaisilla lisävarusteilla, kuten Apple TV tai Chromecast-tyyppisillä suoratoistolaitteilla. Lähes jokaisessa modernissa televisiossa on kuitenkin verkkoselain. PWA voisi olla yksi ratkaisu televisioden sovellukseksi. Jokaiselle televisiolle voitaisiin järkevästi kehittää sovelluksia hyödyntämällä selainta.

Samalla tavalla PWA-sovelluksia voisi hyödyntää autoissa, tai muissa kodin esineiden internet-tyyppisissä laitteissa. Laitevalmistajien yhteistyö olisi tärkeää, koska mikään ekosysteemi tai yritys ei voi tehdä kaikkea itse. Myös käyttöliittymät ja käyttökokemus kulkevat käsi kädessä suorituskyvyn ja laitealustojen kanssa. PWA:n käyttömahdollisuudet ovat tulevaisuudessa lähes rajattomat, mikäli laitevalmistajat ja yritykset ymmärtävät tehdä yhteistyötä ja omaksua uuden teknologian.

Malavoltan tekemässä tutkimuksessa [17] oli mitattu PWA-sovellusten virrankulutusta. Työssä sanottiin satunnaisten muuttujien saattaneen vaikuttaa testituloksiin. Esimerkiksi puhelimen taustalla toimivat sovellukset tai palvelut saattoivat lisätä virrankulutusta satunnaisesti. Lisäksi laitemäärä oli rajallinen ja testiolosuhteet eivät myöskään välttämättä olleet parhaat mahdolliset. Tulevaa tutkimusta kannattaisi jatkaa virrankulutuksen tutkimisella. Miten natiivisovellusten virrankulutus eroaa PWA-sovelluksista jos testiolosuhteita ja laitemäärää parannetaan? Onko PWA-sovellusten virrankulutuksen lisääntyminen merkittävää ja miten se näkyy käyttäjälle? Onko virrankulutuksen muutoksilla vaikutusta teknologiaa valittaessa? Muita mielenkiintoisia suuntia tutkimuksen jatkamiselle olisivat tapaustutkimukset myös muista PWA-sovellus muunnoksista. Tässä työssä esiteltiin Tinderin ja Uberin PWA-sovellusten tapaustutkimusten tulokset, joissa kummassakin tapauksessa sovellusten käyttö lisääntyi PWA-sovelluksena. Sovellusten käytöstä pitäisi saada lisää tietoa. Käyttävätkö käyttäjät mielummin natiivisovelluksia, vai PWA-sovelluksia ja mistä käyttö johtuu? Onko syynä suorituskyky vai sovelluksen löydettävyys?

7 Yhteenveto

Kirjallisuuskatsauksen perusteella PWA-sovelluksista ei ole vielä tehty paljon tieteellistä tutkimusta. PWA-sovellukset ovat uusi teknologia ja sitä on kokeiltu vasta muutamissa suurissa yrityksissä kuten: Tinder ja Uber. Näiden tapaustutkimusten perusteella PWA-sovellusten rakentaminen on järkevää kustannusten ja tehokkuuden vuoksi. PWA-sovellusten käyttö on nostanut käyttäjämääriä sovellusten parissa. Sovellusten latausajat ovat pienentyneet Service Workerin välimuistin hyödyntämisen myötä ja käyttöaste on kasvanut, kun käyttäjän ei tarvitse odottaa sovelluksen latautumista. Service Workerin käyttö on myös mahdollistanut sovelluksen pikakuvakkeen asentamisen suoraan verkkosivulta ja ilmoitusten näyttämisen mobiililaitteella.

PWA-sovellusten rakentaminen on järkevää tilanteissa, joissa ei ole tarvetta natiivisovellusten kaltaiseen suorituskykyyn. PWA-sovellukset ovat myös hyvä keino kustannusten pienentämiseen. Sovellus kehitetään vain kerran ja se toimii kaikilla laitteilla kaikkialla missä on nykyaikainen verkkoselain: työasemilla ja mobiililaitteilla. PWA-sovellukset saattavat olla myös ratkaisu tulevaisuudessa älytelevisioiden ja autojen sovelluksiksi.

TK1: Kuinka paljon nopeampi Service Workerin välimuisti on kuin verkkosivu, joka ei hyödynnä Service Workerin välimuistia?

Googlen tapaustutkimuksen tulokset osoittavat, että Service Workerin välimuistin käyttö nopeuttaa uusiutuvaa sivulatausta noin kaksi kolmasosaa. Sivusto, joka käyttää Service Workeria, latautuu kolmasosassa ajasta kuin sivusto, joka ei ole tallettanut mitään välimuistiin. Myös tämän työn tutkimustulokset tukevat väitettä. Service Workerin käyttö oli kaikissa lähtötilanteissa nopeampaa. Uudet sivulataukset välimuistista olivat vain kolmasosan siitä, mitä sivulataukset ilman välimuistia. Sivulataukset olivat nopeampia molemmissa testitapauksissa. Service Workerin välimuistin käyttö on siis huomattavasti nopeampaa kuin ilman.

TK2: Miten olemassa oleva verkkosivu voidaan muokata PWA-sovellukseksi?

Olemassaoleva verkkosivusto voidaan muokata PWA-sovellukseksi noudattamalla Googlen tarjoamaa tarkastuslistaa. Sivustolle pitää luoda vähintään Web App Manifest, sovelluskuvake, Service Worker ja HTTPS-yhteys. Kun nämä vähimmäisvaatimukset on täytetty, tulee sovelluksesta PWA-sovellus. Huomion arvoista on se, että Service Workerin ei tarvitse vähimmäisvaatimuksissa tehdä mitään. Pelkkä Service Workerin rekisteröinti selaimessa riittää. PWA-sovellusta voidaan myöhemmin laajentaa tukemaan Service Workerin välimuistikäyttöä ja näyttää sovellusilmoituksia. Lisäksi hyvän PWA-sovelluksen mittareita ovat: responsiivisuus, verkkoyhteydetön tuki, tuoreus, turvallisuus, löydettävyys, sitouttava ja asennettava.

TK3: Minkälaisia hyötyjä tai haittoja saadaan käyttämällä PWA-sovellusta?

PWA-sovellus teknologian takia ei tarvitse erikseen kehittää samaa sovellusta mobiililaitteelle natiivikielellä. Yksi ja sama sovellus käy kaikille laitteille. PWA-sovellusta ei tarvitse julkaista sovelluskaupoissa ja maksaa niihin liittyviä lisenssimaksuja.

Myöskään sovelluskauppojen hyväksyntäprosesseja tai käyttöehtoja ei tarvitse hyväksyä. PWA-sovellus voidaan asentaa suoraan verkkosivulta.

PWA-sovellusteknologian avulla käyttäjälle voidaan lähettää sovellusilmoituksia, vaikka sovellus olisi suljettuna. Service Workerin välimuistia hyödyntämällä sovelluksen latausaika saadaan karkeasti kolmasosaan alkuperäisestä. PWA-sovellus tuntuu ja näyttää natiivisovellukselta käyttäjän laitteella. Asennettu PWA-sovellus ei myöskään vie tilaa käyttäjän laitteelta, koska tiedot voidaan hakea suoraan verkosta. Käyttäjän mobiililaitteella on tallennettuna vain sovelluskuvake sekä välimuistissa olevat asiat. Käyttäjän ei ole pakko asentaa tai hyväksyä käyttöehtoja sovellusta käytettäessä. Käyttäjän ei myöskään ole pakko asentaa mitään ellei halua. PWA-sovellusta on mahdollista käyttää myös verkkoyhteydettömässä tilassa.

PWA-sovelluksen haittapuoliin kuuluu lisääntynyt työmäärä, mikäli olemassa oleva verkkosivu halutaan muokata PWA-sovellukseksi. Lisää työtä aiheuttavat kaikki toimenpiteet, jotka lasketaan PWA-sovelluksen vaatimuksiksi ja hyviksi käytännöiksi, mikäli niitä ei ole ennestään toteutettuna. Lisää työtä voi aiheuttaa myös sovelluskuvakkeen suunnittelu ja tallentaminen kaikkiin eri kokoihin. Myös Service Workerin välimuistin käyttö saattaa viedä liikaa muistia, mikäli strategiaa ei ole mietitty kunnolla. Sovelluksen sitouttamisen kannalta tulee miettiä, millaisia sovellusilmoituksia näytetään ja milloin. Väärät ilmoitukset väärään aikaan saattavat vähentää sovelluksen käyttöä ja kiinnostusta sitä kohtaan.

Lähteet

1 G. R. Abhi Gambhir, "Analysis of cache in service worker and performance scoring of progressive web application," tech. rep., ASET - CSE Amity University Uttar Pradesh, Noida, Paris, France, 2018. [Julkaistu 22-23.7.2018][Viitattu 11.10.2018].

2 A. Russell, "Progressive web apps: Escaping tabs without losing our soul." https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955. [Julkaistu 10.8.2015][Viitattu 11.10.2018].

3 A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," in *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)*, pp. 344–351, 2017.

4 S. S. Tandel and A. Jamadar, "Impact of progressive web apps on web app development," *Department of Computer Engineering, Smt.Indira Gandhi College, Navi Mumbai(Ghansoli), Maharashtra, India*, 2018.

5 S. Khalaf, "Seven years into the mobile revolution: Content is king... again." https://yahoodevelopers.tumblr.com/post/127636051988/seven-years-into-the-mobile-revolution-content-is. [Julkaistu 26.8.2015][Viitattu 11.10.2018].

6 S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*, pp. 213–220, ACM, 2013.

7 H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with md 2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 526–533, ACM, 2013.

8 S. Kapoor, "Progressive web apps 101: the what, why and how." https://medium.freecodecamp.org/progressive-web-apps-101-the-what-why-and-how-4aa5e9065ac2. [Julkaistu 20.7.2018][Viitattu 11.10.2018].

9 A. Gazdecki, "Why progressive web apps will replace native mobile apps." https://www.forbes.com/sites/forbestechcouncil/2018/03/09/why-progressive-web-apps-will-replace-native-mobile-apps/. [Julkaistu 9.3.2018][Viitattu 25.10.2018].

10 S. Perez, "Majority of u.s. consumers still download zero apps per month, says comscore." https://techcrunch.com/2017/08/25/majority-of-u-s-consumers-still-download-zero-apps-per-month-says-comscore/. [Julkaistu 2017][Viitattu 25.10.2018].

11 T. Ater, *Building progressive web apps: bringing the power of native to the browser*. "O'Reilly Media, Inc.", 2017.

12 F. Angelini, "The mobile economy digital landscape: la domanda." https://www.comscore.com/Insights/Presentations-and-Whitepapers/2016/The-Mobile-Economy-Digital-Landscape. [Julkaistu 18.11.2016][Viitattu 15.03.2019].

13 C. driven list of stat, "Pwa stats." https://www.pwastats.com/. [Viitattu 06.01.2019].

14 A. Osmani, "A pinterest progressive web app performance case study." https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154. [Julkaistu 29.11.2017][Viitattu 12.11.2018].

15 G. Developers, "Progressive web app checklist." https://developers.google.com/web/progressive-web-apps/checklist. [Julkaistu 24.7.2018][Viitattu 25.10.2018].

16 M. Hiltunen, "Creating multiplatform experiences with progressive web apps," *Metropolia University of Applied Sciences Bachelor of Engineering Information and Communication Technology*, 2018.

17 I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirović, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 35–45, IEEE Press, 2017.

18 P. Walton, "Measuring the real-world performance impact of service workers." https://developers.google.com/web/showcase/2016/service-worker-perf. [Julkaistu 7.2016][Päivitetty 2.7.2018][Viitattu 02.01.2019].

19 J. Archibald, "The offline cookbook." https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/. [Viitattu 15.03.2019].

20 E. G. Philip Walton, "Building faster, more resilient apps with service worker (chrome dev summit 2018)." https://youtu.be/25aCD5XL1Jk. [Julkaistu 12.11.2018][Viitattu 04.01.2019].

21 G. Developers, "Introduction to push notifications." https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications. [Julkaistu 2018][Viitattu 25.10.2018].

22 "Service workers." https://caniuse.com/#search=service%20worker. [Viitattu 15.03.2019].

23 C. Love, "What is the service worker cache storage limit? how much your progressive web app (pwa) can store." https://love2dev.com/blog/what-is-the-service-worker-cache-storage-limit/. [Viitattu 25.01.2019].

24 L. Eisworth, "Progressive web apps 101: the what, why and how." https://www.sangfroidwebdesign.com/search-engine-optimization-seo/google-https-ranking/. [Julkaistu 28.3.2018][Viitattu 11.10.2018].

25 B. Frankston, "Progressive web apps [bits versus electrons]," *IEEE Consumer Electronics Magazine*, vol. 7, pp. 106–117, March 2018.

26 J. von der Assen, "A progressive web app (pwa)-based mobile wallet for bazo," *Bachelor Thesis, University of Zurich*, 2018.

27 K. Behl and G. Raj, "Architectural pattern of progressive web and background synchronization," in *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pp. 366–371, June 2018.

28 A. Osmani, "The app shell model." https://developers.google.com/web/fundamentals/architecture/app-shell/. [Viitattu 08.03.2019].

29 Mozilla, "Web app manifest." https://developer.mozilla.org/en-US/docs/Web/Manifest. [Julkaistu 2018][Viitattu 11.10.2018].

30 A. Charland and B. Leroux, "Mobile application development: web vs. native," *Queue*, vol. 9, no. 4, p. 20, 2011.

31 L. Tung, "Chrome desktop apps move to android, ios with apache cordova." https://www.zdnet.com/article/chrome-desktop-apps-move-to-android-ios-with-apache-cordova/. [Julkaistu 29.1.2014][Viitattu 11.11.2018].

32 A. Cordova, "Architecture." https://cordova.apache.org/docs/en/latest/guide/overview/. [Julkaistu, ei tiedossa][Viitattu 11.11.2018].

33 S. Bosnic, I. Papp, and S. Novak, "The development of hybrid mobile applications with apache cordova," in *Telecommunications Forum (TELFOR), 2016 24th*, pp. 1–4, IEEE, 2016.

34 C. P. Team, "Crosswalk 23 to be the last crosswalk release." https://crosswalk-project.org/blog/crosswalk-final-release.html. [Julkaistu 2017][Viitattu 11.11.2018].

35 Facebook, "React native." https://facebook.github.io/react-native/. [Julkaistu ei tiedossa][Viitattu 11.11.2018].

36 S. Aggarwal, "How react native works." http://www.discoversdk.com/blog/how-react-native-works. [Julkaistu 13.06.2017][Viitattu 11.11.2018].

37 N. Serrano, J. Hernantes, and G. Gallardo, "Mobile web apps," *IEEE Software*, vol. 30, pp. 22–27, Sep. 2013.

38 M. Sojka, "Platform choice: Android vs ios." https://www.ready4s.com/blog/android-vs-ios-comparing-ui-design. [Julkaistu 23.09.2018][Viitattu 16.03.2019].

39 "Webview for android." https://developer.chrome.com/multidevice/webview/overview. [Viitattu 16.03.2019].

40 N. Pande, A. Somani, S. Prasad Samal, and V. Kakkirala, "Enhanced web application and browsing performance through service-worker infusion framework," in *2018 IEEE International Conference on Web Services (ICWS)*, pp. 195–202, July 2018.

41 Yanzi, "Sensors and gateways." https://www.yanzi.se/solution/. [Julkaistu 2018][Viitattu 11.10.2018].

42 R. Wieringa, "Design science methodology: principles and practice," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 493–494, IEEE, 2010.

43 G. Developers, "Lighthouse." https://developers.google.com/web/tools/lighthouse/. [Viitattu 28.12.2018].

44 G. Developers, "Puppeteer." https://developers.google.com/web/tools/puppeteer/. [Viitattu 26.01.2019].

45 N. Jain, A. Bhansali, and D. Mehta, "Angularjs: A modern mvc framework in javascript," *Journal of Global Research in Computer Science*, vol. 5, no. 12, pp. 17–23, 2015.

46 J. Deacon, "Model-view-controller (mvc) architecture," *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

47 M. A. Jadhav, B. R. Sawant, and A. Deshmukh, "Single page application using angularjs," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, pp. 2876–2879, 2015.

48 J. Cryer, *Pro Grunt. js*. Apress, 2015.

49 K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The adoption of javascript linters in practice: A case study on eslint," *IEEE Transactions on Software Engineering*, 2018.

50 J. K. Sebastian McKenzie, Christoph Nakazawa, "Yarn: A new package manager for javascript." https://code.fb.com/web/yarn-a-new-package-manager-for-javascript/. [Julkaistu 11.10.2016][Viitattu 08.03.2019].

51 D. Mazinanian and N. Tsantalis, "An empirical study on the use of css preprocessors," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 168–178, March 2016.

52 T. Maynard, *Getting Started with Gulp–Second Edition*. Packt Publishing Ltd, 2017.

53 V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce, "Twitter bootstrap and angularjs: Frontend frameworks to expedite science gateway development," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–1, IEEE, 2013.

54 V. Subramanian, "Modularization and webpack," in *Pro MERN Stack*, pp. 115–150, Springer, 2017.

55 G. Developers, "Workbox cli." https://developers.google.com/web/tools/workbox/modules/workbox-cli. [Julkaistu 2018][Viitattu 25.10.2018].